Vulnerability Evolution and the Promise of Automated Gatekeeping in Open-Source Software

Seyed Ali Akhavani[1*], Behzad Ousat[2*], Selcuk Uluagac[2], and Amin Kharraz[2]

[1] Northeastern University, Boston, MA, USA
sadatakhavani.s@northeastern.edu
[2] Florida International University, Miami, FL, USA
{bousat,mkharraz,suluagac}@fiu.edu

Abstract. Open-source software (OSS) has become increasingly more popular across different domains, yet the security mechanisms governing it have not scaled with its growth. This paper presents a longitudinal study of 30,572 unique vulnerability reports spanning 10 programming languages and eight package managers from 2017 to late 2025. Our analysis exposes a fundamental imbalance: reported vulnerable packages are growing at 91.5% annually, nearly four times the 26% growth rate of total packages, while maintainer remediation capacity has remained roughly flat at 1,650 deployed fixes per year since 2022. This mismatch has resulted in a growing backlog of unresolved vulnerabilities. We also examine vulnerability distributions across ecosystems and find that only six CWE types account for over 50% of all reports across ecosystems, and that intentional supply chain attacks through embedded malicious code are heavily concentrated in NPM and PyPI, which together account for over 98% of all such incidents.

This imbalance in maintainer capacity, combined with the concentration of risk in a small set of weakness patterns, motivates the exploration of large language models (LLMs) as automated defenses within CI/CD pipelines. We evaluate 10 state-of-the-art LLMs on 582 pairs of real-world vulnerable and patched package versions covering 1,434 advisories. Our results show that no single model resolves the trade-off between precision, recall, and latency. `GPT-5.1` achieves the strongest balanced performance (62.8% F_1), while reasoning-intensive models such as `DeepSeek-R1` incur considerably higher inference costs (∼50s), and high-recall models like `LLaMA-2` generate excessive false positives. These findings support a hybrid defense pipeline that balances recall and precision, combining efficient first-pass screening with more precise downstream analysis for validation and classification. Such designs offer a practical path to narrowing the growing gap between vulnerability discovery and remediation.

Keywords: Open Source Security · Supply Chain Security · AI for Security.

[*] These authors contributed equally to this work.

1 Introduction

The OSS landscape has become the backbone of modern digital infrastructure, powering industries ranging from aerospace and energy to finance and healthcare [16,24,13,65,66,61]. The open and decentralized nature of OSS, while promoting collaboration and innovation, also creates opportunities for adversaries to exploit vulnerabilities [64], compromise supply chains [14,59], and introduce malicious code into widely-used libraries [55,47,9]. Government entities and regulators have responded to these risks with urgent mandates for stronger security. Initiatives like the U.S. *Securing Open Source Software Act* [62,60] and CISA's *Open Source Security Roadmap* [15] now establish frameworks to protect open-source dependencies. These measures underscore the critical need to address dependency visibility and large-scale vulnerability management in modern infrastructure.

The central premise of this paper is that the scale of open-source adoption has outpaced our ability to secure it. While previous studies have examined specific ecosystems such as NPM or PyPI in isolation, a comprehensive, longitudinal understanding of how vulnerability patterns are evolving across the broader software supply chain is missing. To address this, we constructed a comprehensive dataset comprising 30,572 vulnerability reports affecting 16,802 packages across 10 programming languages, C, C++, PHP, Rust, JavaScript (Node.js), Go, Java, .NET, Python, and Ruby, across eight distinct package managers (PyPI, NPM, Packagist, Crates, Go, Maven, NuGet, and RubyGems), all of which are foundational to software and web development [10,3], from 2017 to late 2025.

Our analysis reveals three principal findings. First, **despite rapid growth of advisory reports, the results indicate a growing mismatch between ecosystem expansion, vulnerability discovery, and effective remediation.** Reported vulnerable packages are growing at 91.5% annually, nearly four times the 26% growth rate of total packages across ecosystems. Yet the number of deployed fixes has stabilized at roughly 1,650 per year since 2022, yielding a fix-to-report ratio of approximately 0.44. Between 2022 and 2024 alone, ecosystems accumulated over 7,000 unresolved vulnerabilities, corresponding to an average annual deficit of approximately 2,500. Vulnerability lifespan analysis shows that this surge is substantially driven by pre-existing defects, with average lifespans peaking at 1,988 days in 2022 before declining to 1,475 days by 2025. The open-source community is constrained to address a fraction of reported issues while the backlog of unpatched vulnerabilities continues to grow.

Second, **the distribution of vulnerabilities across weakness classes is heavily skewed.** Only six CWEs account for over 50% of all reported flaws, and in some ecosystems a single CWE type accounts for more than 60% of all issues. This concentration is most visible in the rise of intentional supply chain attacks: malicious packages classified under CWE-506 (Embedded Malicious Code) now account for 67% of all reports in NPM and 14% in PyPI. Unlike traditional software bugs, these are deliberate injections of malware, often utilizing tactics like typosquatting and dependency confusion to compromise users.

Third, **the concentration of vulnerabilities within a small set of recurring patterns, combined with the capped remediation capacity identified above, motivates the use of large language models (LLMs) as automated gatekeepers within CI/CD pipelines.** We evaluated a suite of state-of-the-art models, including `GPT-5.1`, `GPT-4o`, `DeepSeek-R1`, and seven other open-weight models on a curated dataset of 582 pairs of real-world vulnerable and patched versions from unique packages covering 1,434 advisory reports across 10 programming language ecosystems. We find that `GPT-5.1` achieves the strongest balanced performance (62.8% F_1), demonstrating robust performance across diverse ecosystems and remaining comparable to recent approaches reported in the literature, while other models face trade-offs between recall and precision, and reasoning-heavy models introduce high latency (up to 50 seconds per sample). We also uncover a systematic prevalence bias in CWE classification, where models frequently predict CWE-79 (XSS) regardless of the actual vulnerability type.

In summary, this paper makes the following contributions:

- **Longitudinal measurement of the remediation gap.** We quantify the widening disparity between vulnerability discovery and maintainer patching capacity across 10 programming languages over nine years, demonstrating that the open-source community's ability to deploy fixes has not scaled with the volume of incoming reports.
- **Cross-ecosystem vulnerability characterization.** We provide a comparative analysis of universal vulnerability patterns, accounting for most of the reported vulnerabilities across ecosystems. The analysis also uncovers platform-specific patterns, from the dominance of intentional supply chain attacks in NPM to memory safety issues concentrated in C/C++.
- **Empirical evaluation of LLMs as CI/CD gatekeepers.** We design and execute a diff-centric evaluation of 10 LLM architectures on real-world vulnerable-patched version pairs, revealing critical trade-offs between detection accuracy, CWE classification fidelity, and inference cost, and identifying a prevalence bias in some CWE predictions.
- **Open dataset.** We release our unified dataset of normalized vulnerability reports, package metadata, and code diffs to support reproducible research in automated software security. [3]

2 Background and Related Work

This section outlines the different aspects of OSS security research regarding vulnerability detection and automated defenses. We also define the vulnerability disclosure process in the appendix A.4. Table 1 summarizes how our work extends the scope and scale of existing literature.

2.1 Ecosystem Security and Dependency Analysis

Prior research has extensively mapped the structural evolution of open-source dependency networks.

[3] `https://github.com/fiu-seclab/oss-security`

Table 1. Overview of research paper details on package vulnerability and dependency network analysis.

Category	Reference	Total Vulnerabilities	Total Packages	Evaluation Period	Package Managers									
					PyPi	NPM	RubyGems	Maven	Crates	Packagist	NuGet	Go	CPAN	CRAN
Package Vulnerability	[6]	1,396	2,224	2006-20	✓									
	[73]	609	-	2011-18		✓								
	[22]	339	1M	n/a	✓	✓	✓							
	[69]	5,916	958,547	NA-2022	✓	✓								
Dependency Network	[18]	0	449,518	2011-16		✓	✓							✓
	[20]	0	830,000	2012-17		✓	✓		✓	✓	✓		✓	✓
	[31]	0	253,896	2005-16		✓	✓		✓					
Dependency Vulnerability	[19]	400	610,000	2012-17		✓								
	[70]	2,874	867,290	2011-20		✓	✓							
	[46]	n/a	200	n/a				✓						
Package Vulnerability	Our Study	**30,572**	**16,802**	**2017-25**	✓	✓	✓	✓	✓	✓	✓	✓		

Decan et al. [18,20] and Kikas et al. [31] analyzed dependency graphs across major platforms, including NPM, RubyGems, and Crates, to identify structural complexities and the growing risks of transitive dependencies. Building on this structural understanding, several studies have empirically examined how vulnerabilities propagate through these networks. Decan et al. [19] and Zerouali et al. [70] investigated the impact of security reports in NPM and RubyGems, tracking how flaws affect both direct and transitive dependents. Similarly, Pashchenko et al. [46] focused on the Maven ecosystem, proposing methods to distinguish between potential exposure and the actual impact of vulnerable dependencies. More recently, research has shifted toward identifying specific security trends and developer practices within these ecosystems. Alfadel et al. [6] found that Python packages often introduce vulnerabilities in early releases that persist throughout their history, while Zimmermann et al. [73] highlighted the implicit trust model in NPM that creates a massive attack surface. Work by Zahan et al. [69,68] further discovered a correlation between community adherence to security guidelines (such as using automated tools) and a lower prevalence of reported flaws. This work expands our prior work on OSS vulnerabilities [5] by analyzing maintainer remediation capacity and evaluating LLMs as a suggested defense layer.

Our study diverges from this prior work by moving beyond dependency graph analysis to focus on a longitudinal assessment of vulnerabilities across a much broader scope. While previous studies typically examined isolated ecosystems, our work analyzes 30,572 vulnerability reports across 16,802 packages in 10 programming languages. This dataset is five times larger than those in prior studies and uniquely includes an analysis of C/C++ package vulnerabilities, which is a significant gap in the existing literature. Prior work is summarized in Table 1 to illustrate the existing research focus and demonstrate our contributions.

2.2 LLMs for Vulnerability Detection

The integration of LLMs into vulnerability detection workflows marks a shift from purely structural analysis toward semantic-aware reasoning [30]. Unlike traditional deep learning systems that typically operate as black-box binary classifiers, LLMs leverage transformer-based self-attention to model complex language patterns and long-range dependencies in source code. Recent research has increasingly explored context-aware frameworks that hybridize LLMs with traditional program analysis to mitigate the limited context windows of current models. Several comparative benchmarks have been proposed to evaluate LLM performance on real-world code. SecVulEval [4] introduces a fine-grained dataset of statement-level C/C++ vulnerabilities, showing that even state-of-the-art models struggle on realistic detection tasks. In another study [38], the authors conduct a comprehensive study across zero-shot and few-shot settings, highlighting how model family, context window, quantization, and architectural choices influence vulnerability detection in Java and C/C++. Yildiz et al. [67] further demonstrate that agent-based frameworks using thought–action–observation reasoning can outperform standalone LLMs, though substantial limitations remain. Collectively, these studies emphasize the importance of rich context and interprocedural reasoning.

In parallel, new techniques have been developed to improve LLM grounding and security-specific reasoning. Specification-guided methods extract security constraints from historical patches to counteract reliance on surface patterns [72]. Retrieval-augmented generation has also been applied through systems such as Vul-RAG [21], which construct vulnerability knowledge bases to guide inference and significantly improve detection accuracy. Moreover, the LLMxCPG framework [36] integrates Code Property Graphs with LLMs to identify multi-function vulnerabilities using graph-guided slicing, achieving F_1-score gains of up to 40% over prior deep learning baselines. Unified evaluation efforts such as LLM4Vuln [53] seek to disentangle intrinsic model capabilities from external aids like retrieval and prompting, enabling more systematic comparisons across languages and vulnerability classes. These works point toward an emerging class of hybrid systems that combine LLMs with structured analysis, retrieval, and agentic components. However, most evaluate detection performance in isolation and do not examine how such techniques integrate into real-world continuous integration and deployment pipelines.

In the following, we formulate three research questions that examine vulnerability evolution, ecosystem-specific patterns, and the practical viability of emerging automated defenses.

3 Research Questions

Our analysis is driven by three research questions designed to provide actionable findings for developers and security practitioners.

RQ1: How does the exponential growth of reported vulnerabilities contrast with the patching capacity of open-source maintainers? This

question examines the sustainability of open-source security by quantifying the disparity between vulnerability discovery rates, ecosystem package growth, and remediation throughput. First, we compare the annual growth rate of reported vulnerabilities against the growth rate of packages across major ecosystems, assessing whether security oversight is keeping pace with ecosystem growth. Second, we analyze vulnerability lifespans and the annual volume of deployed patches to determine whether maintainers possess sufficient capacity to remediate the growing backlog of disclosed issues. Together, these analyses reveal whether the open-source community is effectively managing its security obligations or falling progressively behind.

RQ2: What ecosystem-specific vulnerability patterns and attack vectors emerge, particularly with respect to malicious packages? Different ecosystems exhibit distinct vulnerability profiles shaped by their design principles, language features, and developer practices. This question explores the distribution of CWEs across platforms to identify which threats are universal and which are specific to individual ecosystems. We further investigate whether the nature of reported vulnerabilities has shifted over time from unintentional implementation errors toward intentional supply chain abuse, and if so, which deployment patterns, targeting strategies, and attack vectors characterize these deliberate threats.

RQ3: How effective are LLMs as automated gatekeepers for vulnerability detection within the CI/CD pipeline? A significant challenge in open-source security is the limited availability of expert reviewers capable of performing in-depth security analysis at scale, and the difficulty of performing a real-time scan for vulnerability detection. This question evaluates the potential of LLMs to help with this constraint by providing automated support for vulnerability detection during the package publication process. Given the growth and complexity of the OSS projects, we investigate how effectively various model architectures can identify prevalent CWEs within code changes before they are released to the public. By simulating the deployment of these models within continuous integration and delivery (CI/CD) workflows, we measure their performance and characterize their failures across multiple programming languages in order to determine their practical viability as proactive defenses for open-source registries.

4 Data Collection

The pipeline for this research integrates two primary categories of data: longitudinal vulnerability reports from established security databases, including GitHub Advisory Database [1] and Snyk Vulnerability Database [48] as well as a curated dataset of source code modifications designed for evaluating automated detection tools. We aggregate the reported vulnerabilities from the two sources to analyze vulnerability history and trends. To evaluate the effectiveness of defenses, we supplement this with metadata such as the list of published versions of a package from Libraries.io [37] and repository information and metadata

from GitHub. This integrated view enables us both to trace the evolution of vulnerabilities in widely used package registries and to experimentally assess the capability of LLMs to identify flaws at the point of introduction. The methodologies underlying our LLM-based evaluations, including model selection and experimental design, are described in detail in subsequent sections.

4.1 Vulnerability Reports and Package Information

Vulnerability Databases. We aggregate vulnerability reports from two established sources. The GitHub Advisory Database [1] uses the Open Source Vulnerability format (OSV) [2], providing structured reports that include package name, ecosystem, severity, affected versions, remediation steps, and CVE/CWE identifiers. GitHub maintains both reviewed advisories (over 25K reports with verified ecosystem and package mappings) and unreviewed advisories (over 288K reports). We restrict our collection to reviewed advisories to ensure data reliability, extracting 24,500 records across eight ecosystems. The Snyk Vulnerability Database [48] aggregates data from its own disclosures, the National Vulnerability Database, community contributions, and proprietary research. Prior work has utilized Snyk for code analysis [33,54], Docker vulnerability analysis [32], and vulnerability mitigation tools [42]. We collected 23,709 reports from Snyk. Table 2 presents detailed breakdowns by platform from both sources.

Longitudinal Data from Package Managers. A key aspect of our study relies on historical package ecosystem data. We used Libraries.io [37], which provides current package counts across major platforms but does not retain historical records. To overcome this, we retrieved archived snapshots from the Wayback Machine [29]. Using these snapshots, we collected package counts for each ecosystem from 2017 to late 2025, sampling twice per year (January and July) to track growth trends. We also collected the complete list of available versions for any of the reported packages from Libraries.io's website. We were able to successfully collect the version number and publication date of 16,618 unique packages. We utilize this data to assign an introduction date (i.e., first vulnerable version's release date) and a fix date (i.e., first patched version release date) to the reported vulnerabilities. Note that in most cases, the introduction date differs from the publication date because it takes time for the vulnerability to be discovered and published.

4.2 Vulnerable and Patched Codes

Additionally, we construct a corpus of file changes that either introduce or fix known vulnerabilities. For each vulnerability report, we resolve version ranges (e.g., `[1.0.0, 2.0.0)`) to identify four concrete releases: the first patched version, its preceding vulnerable version, the first vulnerable version, and its preceding safe version. This yields two vulnerable-safe version pairs per advisory. Using the GitHub Compare API and excluding pre-release version tags, we extract modified files between each version pair, forming a *Vulnerable Set* and a *Patched Set*. Each vulnerable-safe version pair provides two data points for evaluation: the vulnerable version labeled true with the corresponding CWE and

Table 2. Overview of vulnerability reports collected from GitHub Advisory and Snyk across ecosystems. The final unified dataset contains 30,572 reports after removing 11,274 duplicate entries and excluding 6,363 reports associated with prohibited CWEs.

Platform	Snyk	GitHub Advisory	Unified Reports	Affected Packages
C/C++	3,376	–	2,329	575
Packagist (PHP)	1,992	4,820	4,313	811
Crates (Rust)	914	1,063	1,037	558
Go	2,904	2,719	2,330	1,397
Maven (Java)	2,516	5,861	5,000	2,269
NPM (Node.js)	7,938	4,340	9,933	8,786
NuGet (.NET)	1,310	714	715	314
PyPI (Python)	2,196	4,073	4,122	1,719
RubyGems (Ruby)	563	910	793	373
Total	23,709	24,500	30,572	16,802

the patched version labeled false. This diff-based approach reflects how vulnerabilities are introduced or patched during the software development lifecycle, enabling assessment of LLMs to flag vulnerable changes at the stage they would realistically enter production. We restrict our analysis to the five most prevalent CWEs per ecosystem (21 CWEs total), yielding 807 unique packages across eight package managers. The dataset is smaller than the total number of advisories due to missing historical releases, unavailable repositories, or incomplete version histories. Our analysis of collected diffs shows substantial size variation, with a median of 900 changed lines and a long tail exceeding 10,000 lines. These large diffs typically result from coarse-grained version jumps with aggregated multiple commits, refactors, or feature additions, rather than isolated vulnerability fixes, making them an over-approximation of the true vulnerability-fixing changes due to aggregation across multiple commits. To ensure feasibility for LLM evaluation while maintaining realistic development conditions, we limit the dataset to diffs with at most 5,000 changed lines ($\sim$32K tokens on average). This filtering removes extreme cases caused by versioning artifacts while preserving representative changes. The final dataset consists of 582 version pairs, producing a total of 1,164 data points and covering 1,434 advisory reports.

4.3 Integrated Dataset Generation

Although the OSV format has been introduced to streamline advisory databases, adoption remains inconsistent. Even among databases that use OSV, multiple formats exist. We unified reports from GitHub Advisory and Snyk by standardizing version formats (using bracket/parenthesis notation) and normalizing fields to: `ADVISORY_ID`, `SOURCE`, `PUBLISH_DATE`, `CWES`, `CVES`, `ECOSYSTEM`, `PACKAGE_NAME`, `VERSION_RANGE`, `REFERENCES`. Duplicate reports were removed by matching on `CVE` or, when unavailable, on `CWE`, `Package`, `Ecosystem`, and `Affected Versions`, eliminating 11,274 duplicates (prioritizing Snyk for richer metadata). We further excluded 6,363 reports associated with CWEs that MITRE has flagged as discouraged or prohibited (e.g., CWE-400, CWE-200) since these CWEs are

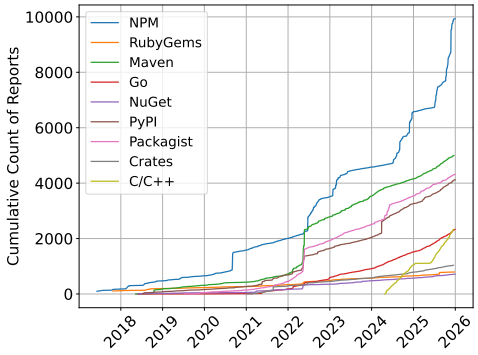

Fig. 1. Number of vulnerabilities reported across ecosystems through the years. Reported vulnerabilities increase in every programming language, with an average of over 500 new vulnerabilities in each ecosystem per year.

considered too broad for actionable mitigation. The full list appears in Appendix A.1.

The final dataset comprises 30,572 vulnerability reports, 467 distinct CWEs, and 16,802 unique affected packages (2017–late 2025), including: (1) package counts per ecosystem, (2) unified vulnerability reports, (3) package metadata with affected versions and CWE/CVE mappings, (4) vulnerability fix dates and counts, and (5) vulnerable and patched code from hundreds of unique packages. Table 2 summarizes the data. We will maintain and extend this dataset to support ongoing research.

In the next three sections, we aim to answer the research questions raised in Section 3 by doing a large-scale security analysis in OSS. We analyze data from 10 programming languages: Node.js, PHP (Composer), Python, Java, Ruby, Go, Rust, .NET (C#), C, and C++. Eight corresponding package managers are also included in this evaluation: NPM, Packagist, PyPI, Maven, RubyGems, Go, Crates (Cargo), and NuGet.

5 Vulnerability Dynamics and Distribution

This section addresses RQ1 by examining the relationship between open-source ecosystem growth and security outcomes. We first analyze trends in package expansion and vulnerability reporting to quantify the disparity between ecosystem growth and security oversight. We then examine annual patch deployment and vulnerability lifespans to assess whether maintainer remediation capacity is sufficient to address the accumulating volume of disclosed issues.

5.1 Trends in Package Expansion and Vulnerability Reporting

Increase in Total Vulnerabilities Across Languages. We observe that the total number of reported vulnerabilities has increased substantially across all

ecosystems. As shown in Figure 1, vulnerabilities rise across every programming language from 2017 onward, with an average of over 500 new reports per year. This increase is especially pronounced in fast-growing ecosystems such as NPM, Maven, and Packagist, where rapid package expansion coincides with a sharp rise in reported vulnerabilities. In 2022, the number of reported vulnerabilities across different platforms significantly increased, with Maven and Packagist reporting 2,073 and 1,477 vulnerabilities, respectively. This increase can be attributed to several factors, including heightened government and regulatory awareness regarding cybersecurity [62] and high-impact supply chain attacks (such as Log4Shell [28], MoveIt [35,12], and Solarwinds [11] vulnerabilities).

Increase in Total Number of Packages Across Ecosystems. Across all studied platforms, we observe an average of 26% annual increase in the total number of packages. The analysis indicates the growing popularity and widespread adoption of open-source software, especially in ecosystems such as NPM, PyPI, and Maven, since they have been experiencing exponential growth over time. A more detailed analysis of the increase in the number of packages and some exceptions for temporary declines is discussed in the Appendix A.3.

Growth Rate of Vulnerable Packages vs. Total Packages. Building on top of the previous analysis, we assess the evolution of reported vulnerabilities with the growth of package managers over time, focusing on the number of vulnerable packages and the total number of packages. We define growth rates as the year-over-year percentage change in each measure. Our analysis indicates that vulnerability growth has consistently outpaced overall package growth across nearly all ecosystems. On average, the number of packages affected by reported vulnerabilities increases by 91.54% per year, compared to an annual growth rate of 26% for total package counts. Appendix A.5 discusses the growth rate comparison for each platform.

These findings establish the first component of RQ1. Vulnerable package growth (91.5% annually) is substantially outpacing total package growth (26% annually) across nearly all ecosystems. This widening gap raises a critical question addressed in the following section: can maintainers remediate vulnerabilities at a rate sufficient to prevent an accumulating backlog of unpatched issues?

5.2 Remediation Capacity and Vulnerability Lifespan Analysis

To answer the previously mentioned question, we analyze remediation capacity through annual patching throughput and vulnerability lifespan trends. We define vulnerability lifespan as the cumulative duration a vulnerability persists across all affected version ranges before remediation. This metric captures how long defects remain exploitable in production. A formal definition appears in Appendix A.4.

Annual Patch Deployment. We examined the number of patched vulnerabilities annually to assess remediation throughput. As illustrated in Figure 2, fix counts increased from 467 in 2017 to 1,757 in 2022, but subsequently stabilized, averaging approximately 1,650 fixes annually between 2022 and 2025. This

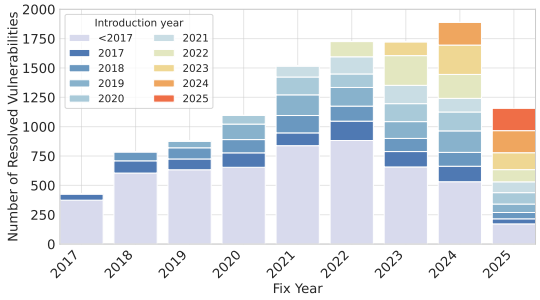

Fig. 2. Number of fixed vulnerabilities based on the introduction year.

stabilized plateau coincides with a rapid increase in vulnerability reports, suggesting that maintainer capacity has reached a practical ceiling. Although our CVE collection window starts in 2017, Figure 2 includes a pre-2017 introduction segment, which reflects vulnerabilities reported from 2017 onward, whose introduction dates trace back to earlier years.

The comparison between fixes and newly reported vulnerabilities (Figure 8 in the Appendix) reveals a persistent and widening gap. Since 2020, the number of patched vulnerabilities has consistently fallen short of reported cases, yielding a fix-to-report ratio of approximately 0.44. After 2021, this ratio remains well below 1, indicating that vulnerabilities are being reported faster than they are resolved. This imbalance highlights a structural limitation in remediation capacity. Despite the surge in reports, the absolute number of fixes remains bounded below ∼2,000 per year. As a result, maintainers appear to prioritize a subset of vulnerabilities, leaving many unresolved and contributing to an accumulating backlog. Between 2022 and 2024 alone, ecosystems accumulated over 7,000 unresolved vulnerabilities, corresponding to an average annual deficit of approximately 2,500.

Vulnerability Lifespan Trends. Moreover, we analyze vulnerability lifespans to assess how long defects persist before remediation. Figure 3 presents trends grouped by the year vulnerabilities were fixed, avoiding right-censoring bias that artificially shortens lifespans for recent code. The average lifespan increased from 1,215 days in 2017 to a peak of 1,988 days in 2022, before declining to 1,475 days in 2025. Despite this decline, the 2025 average remains 21% higher than the 2017 baseline.

The 2022 peak reflects substantial discovery of pre-existing defects, i.e., longstanding vulnerabilities introduced years earlier that were only detected as scanning tooling improved and community awareness increased. Figure 2 provides further context for the subsequent decline. While earlier periods show a broader spread of fixes across older vulnerabilities, recent years exhibit a clear shift toward prioritizing newly introduced issues. In particular, fixes in 2025 are focused on vulnerabilities introduced in 2024 and 2025, with comparatively limited remediation of older vulnerabilities, which lowers the average observed lifespan for that year.

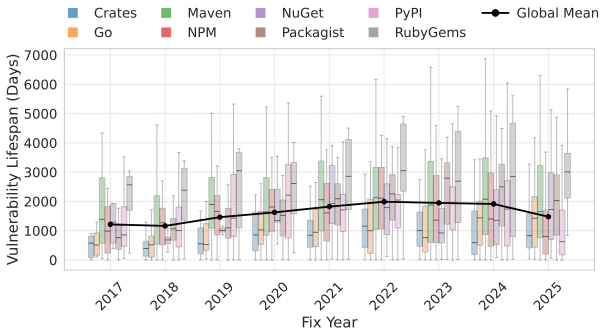

Fig. 3. Vulnerability lifespan trends across platforms. The average lifespan rose by 63.7% from 1,215 days in 2017 to a peak of 1,988 days in 2022, then declined by 25.7% to 1,475 days in 2025, suggesting a recent improvement in remediation.

These findings address RQ1: the exponential growth of reported vulnerabilities has substantially outpaced both ecosystem expansion and maintainer patching capacity, creating a widening remediation gap that poses a significant risk to the software supply chain.

6 Vulnerability Patterns across Ecosystems

In this section, we evaluate CWE trends across the studied platforms by analyzing the presence of reported vulnerabilities in different ecosystems. As mentioned in Section 3, we aim to identify trends in vulnerability distribution for common CWEs and CWEs specific to ecosystems.

6.1 CWEs in Different Ecosystems

Significance of Top CWEs. We analyzed the vulnerability patterns across different ecosystems by investigating the most prevalent vulnerability categories and calculated the number of vulnerabilities originating from the top CWEs. This analysis helps assess whether vulnerabilities in a given platform are concentrated in a small set of CWE types or spread across a broader range. The results (illustrated in Figure 9 in the Appendix) reveal a skewed distribution: in ecosystems such as NPM and Packagist, a small number of CWE types contribute to a large proportion of vulnerabilities. In contrast, platforms like Go and Crates indicate a wider distribution of vulnerability types with less concentration in specific CWEs. Additionally, the analysis suggests that, on average, only six CWEs in different ecosystems account for over 50% of the vulnerabilities. In the following, we analyze the most frequently observed CWEs and explore their characteristics to gain deeper insights into the nature of vulnerabilities.

Popular CWE Types in Different Ecosystems. We analyze the distribution of vulnerabilities across programming languages by identifying both common and ecosystem-specific CWEs. To better understand the existing patterns, we created

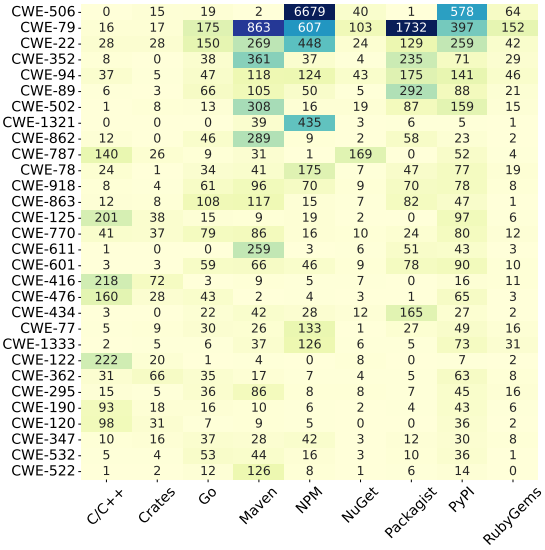

Fig. 4. Heatmap of the number of vulnerabilities from the top 30 CWEs across different platforms. This suggests that a set of CWEs are common among different programming languages, while others are mainly observed in specific ecosystems.

a list of top 30 CWEs across all ecosystems. Figure 4 illustrates a heatmap based on the number of vulnerabilities from the most prevalent CWEs in different platforms. This allows us to investigate security trends while highlighting the unique characteristics of certain ecosystems. In the following, we discuss the insights captured from this analysis.

General Vulnerability Patterns. We observe that multiple CWEs, including CWE-79 (Cross-Site Scripting), CWE-22 (Path Traversal), and CWE-94 (Code Injection) are common in several languages over the years and keep growing at a high rate. These common vulnerabilities emphasize fundamental challenges, specifically in web application security and data handling. Most web development environments face similar challenges, including user-generated content, improper sanitization, and incorrect output encoding. Table 5 in Appendix provides more information for these common CWEs.

Ecosystem-Specific Vulnerability Patterns. Our analysis highlights a set of CWEs that are present in specific ecosystems. For instance, we observe that CWE-1321 (Prototype Pollution) is seen in 435 reports in NPM. This is caused by JavaScript's prototype inheritance model, where objects can inherit properties from other objects (prototypes). Other examples include CWE-416 (Use After Free) and CWE-362 (Race Condition) which are mainly observed in C/C++ and Crates ecosystems due to their memory handling complexities. Table 5 in the Appendix provides examples of specific CWEs in different ecosystems and a brief description for each.

Table 3. Distribution of CWE-506 vulnerabilities across ecosystems. This CWE is mainly reported in NPM and PyPI, with much fewer instances in other ecosystems.

Ecosystem	CWE-506 Count	% of All CWE-506	% of Ecosystem's CWEs
NPM	6,679	90.28%	67.24%
PyPI	578	7.81%	14.02%
RubyGems	64	0.87%	8.07%
NuGet	40	0.54%	5.59%
Go	19	0.26%	0.82%
Crates	15	0.20%	1.45%
Others	3	0.04%	
Total	**7,398**	**100.00%**	**-**

Overall, we observe that specific vulnerability patterns, such as XSS, Path Traversal, and Code Injection are present in multiple ecosystems. However, other classes of vulnerabilities, such as Prototype Pollution or Buffer Overflows, are deeply tied to language characteristics. Each of these vulnerability types potentially requires a different mitigation approach to defend against. Ecosystem-specific trends emphasize the need for custom analysis rules, threat modeling, and education tailored to the unique risks of each development environment. Figure 10 in the Appendix provides the trend of top CWEs in each ecosystem.

6.2 Analysis of Packages with Malicious Intent (CWE-506)

CWE-506 (Embedded Malicious Code) represents an escalating threat in open-source ecosystems. Unlike other vulnerabilities, which often stem from negligence or poor design, these incidents are rooted in intentional abuse. That is, these vulnerabilities emerge from packages deliberately crafted to be exploited. The analysis of the dataset from 2017 to late 2025 shows 7,398 reports classified under CWE-506. Notably, these incidents surged from just 38 reports in 2018 to a staggering 2,952 in 2025. This section explores the distribution of these malicious packages, their targeting strategies, and evolving attack vectors.

Distribution Across Ecosystems. Table 3 presents the distribution of CWE-506 vulnerabilities across different ecosystems. It reports the absolute count of malicious package incidents, each ecosystem's relative share of the total, and the proportion of vulnerabilities attributable to intentional attacks, providing a clear overview of how embedded malicious code is distributed in the software supply chain. Our analysis reveals a high concentration of malicious packages within the NPM and PyPI ecosystems, which together account for more than 98% of all CWE-506 reports. The situation is especially concerning in NPM, where 67% of all reported vulnerabilities stem from packages designed to inflict harm rather than from unintentional bugs.

Several ecosystem-specific factors contribute to this trend. As noted in prior research [73], NPM's low publication barriers, automatic dependency installation, and vast package ecosystem make it particularly attractive for attackers. Bagmar et al. [7] describe how PyPI's packaging model allows malicious packages to propagate easily through dependencies. More recently, Guo et al. [25] found

that PyPI lacks robust behavioral vetting and monitoring, leaving the ecosystem vulnerable to complex supply chain attacks. These systemic flaws across both platforms significantly increase their attack surface.

Attack Vectors. We analyzed the metadata of all 7,398 malicious packages to characterize deployment strategies. Attackers frequently use long package names (>10 characters, 75.6%) and dashes (71.7%) to mimic legitimate naming conventions, and nearly half (49.7%) target all versions using wildcards to maximize exposure, while 27.3% target specific versions for more controlled deployment. We identified three dominant mechanisms used to compromise users:

Typosquatting: Attackers exploit human error by mimicking packages [45]. For example, we observed variants like `lodahs` and `jquery.js`. A single campaign in March 2024 flooded PyPI with over 200 packages mimicking libraries like `tensorflow` (`tensoflouw`) and `pygame` (`pygaqme`). Consistent with this, 11.4% of malicious packages use names closely resembling widely adopted libraries.

Dependency Confusion: This sophisticated vector exploits package resolution priorities by registering public packages with names matching internal private dependencies [8,49]. We identified 1,156 scoped packages using this tactic, confirming attacks against organizations (e.g., `@swiggy-private/analytics`) [51]. Notably, 15.6% of malicious packages use scoped names (e.g., `@org/package`) to impersonate internal corporate components, a tactic closely aligned with dependency confusion.

Installation Hook Exploitation: In NPM, attackers abuse lifecycle scripts (`preinstall`, `postinstall`) to execute arbitrary code immediately upon download [49]. Unlike passive bugs, these hooks compromise systems before the package is even imported. Examples include `axios` [34], which executes the postinstall script that acts as a remote access trojan (RAT).

Even trusted packages can become vectors when compromised, as discussed in Appendix B.2. The high prevalence of these intentional threats, combined with the fact that just six CWE types account for over 50% of all vulnerabilities, suggests a clear opportunity for automated gatekeeping. Consequently, the following section evaluates the efficacy of integrating LLMs into development workflows to identify these high-impact weaknesses before publication.

7 LLMs as Vulnerability Gatekeepers

The concentration of over half of all vulnerabilities within six CWE types, suggests that automated detectors do not need to generalize across each and every category to be useful. They only need to perform well on a small, high-impact set. Moreover, the findings from Section 5.2 suggest that manual remediation has reached an apparent practical limit of approximately 2,000 per year while the backlog of unpatched vulnerabilities continues to grow. Automated detection at the point of code submission could help prevent new vulnerabilities from entering this backlog in the first place, reducing the burden on maintainers who are already unable to keep pace with existing reports.

This section evaluates whether current LLMs can fill that role within CI/CD pipelines. We adopt a diff-centric approach, analyzing the added and removed

Table 4. Comprehensive Overview of Evaluated Models: Properties, Performance, and Cost.

Model	Model Properties				Performance			Cost	
	Provider	Scale	CW	Code-Specialized	Precision	Recall	F_1	Total Tokens (M)	Avg Time (s)
`CodeGemma:7B`	Ollama	Small (7B)	8K	✓	40.1%	59.4%	47.9%	5.90	1.68
`DeepSeek-Coder-V2:16B`	Ollama	Medium (16B)	160K	✓	47.7%	37.3%	41.9%	13.77	4.46
`DeepSeek-R1:70B`	Ollama	Large (70B)	128K	–	53.5%	65.6%	59.0%	13.32	49.90
`Qwen-2.5-Coder:32B`	Ollama	Large (32B)	32K	✓	47.8%	45.4%	46.6%	12.54	5.00
`Phi-3:14B`	Ollama	Medium (14B)	128K	–	52.6%	34.2%	41.4%	14.30	4.30
`LLaMA-2:70B`	Ollama	Large (70B)	4K	–	45.8%	84.9%	59.5%	3.76	6.81
`LLaMA-3.3`	Ollama	Large (70B)	128K	–	39.9%	34.4%	36.9%	12.42	9.55
`Mixtral:8x7B`	Ollama	MoE (8×7B)	32K	–	90.1%	11.0%	19.6%	14.21	2.52
`GPT-4o`	OpenAI	–	–	–	53.2%	69.7%	60.3%	12.85	2.39
`GPT-5.1`	OpenAI	–	–	–	63.9%	61.8%	62.8%	13.67	2.50

lines associated with a version upgrade, which aligns with how code changes enter production in modern development workflows. Prior evaluations [38,4,67] largely rely on function-level or whole-repository snapshots and simplified code snippets, while existing benchmarks such as BigVul [23] and Devign [71] do not capture context-dependent vulnerabilities introduced through evolving codebases. Our evaluation addresses this gap by testing models on real version diffs drawn from the same ecosystems analyzed in prior sections.

7.1 Model Selection and Prompting

Model Catalog. Our experimental setup adopts a provider-agnostic architecture to enable systematic comparison across a diverse set of language models, including proprietary commercial systems and open-weight alternatives deployed locally via Ollama. The evaluated models span general-purpose and code-specialized architectures, a wide range of parameter scales, with different context windows (CW) and architectures. The complete list of models, together with their scale and context-window sizes, is summarized in Table 4.

Diff-Based Prompting. To align with real-world development workflows, we adopt a diff-centric formulation that analyzes only the added and removed lines associated with version updates. Compared to file-level or repository-level inputs, this approach preserves the minimal context required for reasoning about vulnerability introduction and mitigation, while remaining compatible with CI/CD pipelines. The curated version-level diffs may include unrelated changes such as refactoring or feature updates, introducing noise that makes the task more challenging but also more representative of real-world development. Each prompt requests structured JSON output indicating vulnerability presence, CWE classification, and a brief justification. The full prompt appears in Appendix C.

7.2 Vulnerability Presence Analysis

Table 4 summarizes model performance across all evaluated systems. We observe substantial variation across models, indicating that even binary vulnerability detection remains highly sensitive to model choice. `CodeLLaMA` is excluded as it

consistently refused to provide predictions due to safety restrictions. Additional configurations ($temp = 0$, $top_p = 0.1$) yielded minimal impact, so we report results under default settings. Despite the moderate performance observed, the results are broadly comparable to prior work on LLM-based vulnerability detection. For example, recent evaluations report F_1 scores in the range of 57–66% across both function-level settings [38] and repository-level, agent-based approaches [67], mainly focused on Java and C/C++ datasets. In contrast, our diff-based evaluation achieves competitive performance (e.g., $F_1 = 62.8\%$ for `GPT-5.1`) while operating on real-world version changes of 10 different ecosystems. This suggests that diff-based analysis provides a realistic and challenging setting, while still maintaining performance comparable to existing approaches.

Precision vs. Recall. Models exhibit clear differences in operating behavior. Some systems, such as `LLaMA-2:70B`, favor high recall (84.9%) at the cost of low precision (45.8%), making them suitable for exploratory analysis but costly for automated pipelines. `GPT-4o` and `DeepSeek-R1:70B` show a similar tendency toward higher recall (69.7% and 65.6%) with moderate precision. In contrast, `Mixtral-8x7B` is highly selective, achieving very high precision (90.1%) but extremely low recall (11.0%), limiting its effectiveness as a standalone detector while making it suitable for confirmation stages. Among all evaluated models, `GPT-5.1` provides the most balanced performance ($F_1 = 62.8\%$, accuracy = 63.4%), with `GPT-4o` and `DeepSeek-R1:70B` following closely. Overall, models operating in this middle regime offer the most practical utility, as both missed vulnerabilities and excessive false positives impose real-world costs.

Token Utilization and Latency. Beyond detection accuracy, deployment efficiency depends on computational overhead. Token consumption varies substantially across models: lightweight systems such as `LLaMA-2:70B` and `CodeGemma:7B` process fewer tokens, while more verbose models, including `Mixtral-8x7B` and `GPT-5.1`, exceed 13.5M tokens, increasing memory and monetary cost. Latency differences are also noticeable. Open-weight models such as `CodeGemma:7B` and `DeepSeek-Coder-V2:16B` are relatively fast, whereas `DeepSeek-R1:70B` incurs significant delays (50s). On the contrary, proprietary API-based systems such as `GPT-4o` and `GPT-5.1` achieve low latency despite their scale, benefiting from optimized infrastructure. All self-hosted experiments used four NVIDIA RTX 4090 GPUs (24GB each). While the reported times may vary depending on hardware and deployment, these results highlight a fundamental trade-off between detection performance and deployment cost in real-world settings.

7.3 CWE Classification Analysis

While binary vulnerability detection provides a useful first-order signal, the practical utility of LLM-based security analysis depends on a model's ability to identify the *type* of vulnerability. CWE classification is essential to figure out which remediation strategies are applicable. Figure 5 shows confusion matrices for the top 10 CWEs from six different models that highlight key patterns.

Prevalence Bias Toward CWE-79. A dominant trend across nearly all models, particularly `CodeGemma:7B`, `LLaMA-2`, and `DeepSeek-Coder-V2`, is a bias

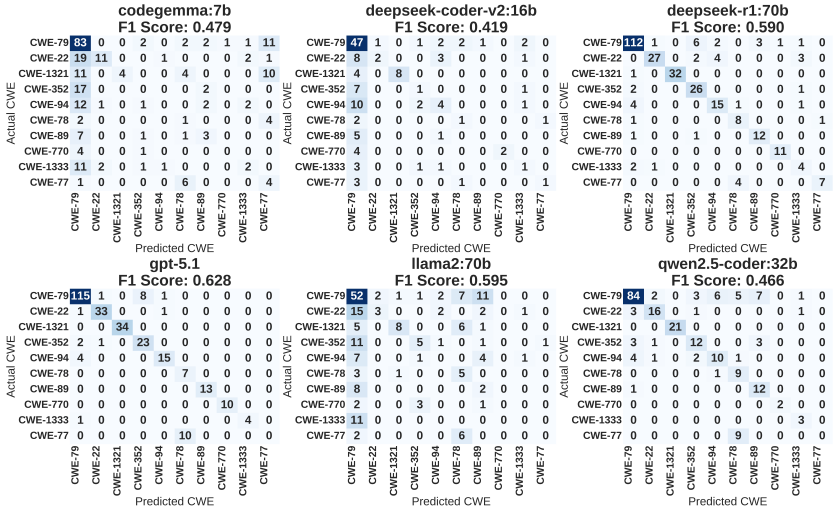

Fig. 5. Confusion Matrix for Actual vs Found CWEs in True Positive Cases. Results indicate a prevalence bias toward CWE-79 and divergent reasoning about CWEs.

toward CWE-79 (Cross-Site Scripting). Vulnerabilities from related injection classes (e.g., CWE-94) or orthogonal categories are often misclassified as XSS, reflecting a *prevalence bias* toward frequent CWEs in their training data.

Divergent Detection Reasoning. Models with similar binary F_1 scores can differ substantially in their ability to correctly classify vulnerability types. For example, `CodeGemma:7B` and `Qwen-2.5-Coder:32B` both achieve ≈ 0.47 F_1 for vulnerability detection, yet `Qwen` exhibits a clearer diagonal structure in its confusion matrix, indicating stronger categorical discrimination, while `CodeGemma:7B` disperses predictions across multiple CWE categories. `GPT-5.1` demonstrates the strongest overall discrimination, with pronounced diagonal dominance across most CWEs. However, even high-performing models struggle with closely related categories, such as authorization pairs (CWE-862 vs. CWE-863) and command injection variants (CWE-77 vs. CWE-78), where distinctions rely on subtle contextual differences rather than surface-level patterns.

Detecting Malicious Packages (CWE-506). The prevalence of CWE-506 (*Embedded Malicious Code*), discussed in Section 6.2, motivates a focused evaluation on this class of threats. Unlike conventional vulnerabilities introduced and patched incrementally, malicious packages often contain backdoors or harmful functionality from the first release, making diff-based analysis unsuitable. Additionally, confirmed malicious packages are removed from repositories to safeguard the users. Therefore, we utilize the publicly available Datadog dataset [17], randomly selecting 600 confirmed malicious packages to match the dataset size used in the main vulnerability detection experiments. Source files were analyzed individually, each submitted to the LLMs with a prompt requesting identification of malicious behavior consistent with CWE-506. Among the evaluated models, `GPT-5.1` (83.5%) and `LLaMA-3.3` (83.3%) achieved the highest detection

rates, while `CodeGemma:7B` (6.7%) performed the worst. `Mixtral-8x7B` (72.9%) and `Qwen-2.5-Coder_32B` (74.4%) also showed strong performance, with other models falling in between. These results highlight that model choice has a substantial impact on the ability to recognize embedded malicious logic in raw source code. Full detection rates for all models are provided in Appendix C.2. While the evaluated models perform well in detecting embedded malicious logic within source files, they struggle with supply-chain attacks introduced through dependency updates. For example, incidents such as `axios@1.14.1` [34] introduce malicious behavior via external dependencies, which is not visible within the analyzed code or diff. As a result, prompt-based analysis alone is insufficient to detect such threats and must be augmented with external threat intelligence (e.g., vulnerability advisories) beyond code-level reasoning.

The results suggest that LLMs offer a promising path toward automated vulnerability detection to help mitigate the growing backlog. While the observed performance metrics are moderate, they are broadly comparable to those reported in prior work on vulnerability detection, and are likely to improve as LLM capabilities continue to advance. Importantly, no single model achieves both high coverage and precision, highlighting the limitations of deploying LLMs as standalone detectors. Instead, practical deployments can utilize multi-stage pipelines that balance these trade-offs, using fast, high-recall models to filter candidate vulnerabilities, followed by slower, high-precision analyzers for validation. Such designs align well with CI/CD workflows and enable scalable, targeted detection of high-impact CWEs before vulnerabilities enter the software supply chain. Finally, it is important to distinguish between detection and remediation. While LLMs show promise in identifying vulnerable patterns, the process of fixing these vulnerabilities remains a fundamentally different challenge that requires deeper semantic analysis and evaluation, and is therefore beyond the scope of this work.

8 Discussion

The Role of Package Managers. Our findings reveal that Go's vulnerability growth has been the highest among the studied ecosystems. However, Go has also made significant strides in security, such as with Go Modules, which replaced GOPATH to improve dependency management by removing deprecated and less-maintained packages. In contrast, Rust's Crates.io is the only ecosystem with a negative growth rate in vulnerable package counts. This may reflect the security nature of Rust or its lower popularity compared to more commonly targeted ecosystems. We further observe that the Go community has actively addressed common issues such as CWE-22 (Path Traversal), which was among the top five vulnerability types in their ecosystem, as seen in the growing family of safe Go libraries [27]. These libraries, such as `SafeText`, `SafeOpen`, and `SafeArchive`, address common security issues in Go, including input validation and path traversal. This suggests that while vulnerabilities in Go have risen, the community has become more proactive, constantly developing defenses and keeping the ecosystem actively maintained. Overall, package managers play a critical role in reducing security risks by encouraging secure defaults and promoting active package maintenance. For example, clearer guidelines for package

deprecation, more secure default configurations, and accurate guidance on package submission could help developers maintain packages more securely.

Limited Usage of Existing Protection Protocols. Despite ongoing efforts to strengthen open-source security through mechanisms such as automated vulnerability scanning and transparent builds like Provenance [26] in the NPM registry, these protections remain underutilized. Although Provenance was introduced over two years ago, it has not gained widespread adoption, even among high-risk packages like `rc`[44] and `COA`[43], which were referenced in Provenance's own launch materials yet still do not enable it. Examining NPM's monthly top 100 most-downloaded packages from `npmleaderboard.org` in February 2026, we found that only 9% had enabled Provenance, and widely used packages such as `express` have not adopted it. This gap between tool availability and use leaves many repositories exposed to preventable attacks, emphasizing the need for stronger incentives, improved defaults, or tighter workflow integration.

9 Conclusion

This paper presents a longitudinal analysis of the open-source security landscape across 10 programming languages, revealing a widening security gap. Packages affected by reported vulnerabilities are surging at an annual rate of 91.54%, significantly outpacing the 26% growth of total packages. More concerning, maintainer patching capacity has plateaued. The number of deployed fixes averaged approximately 1,650 annually from 2022 to 2025, and did not scale with the increasing volume of vulnerability reports. Vulnerability lifespans peaked at 1,988 days in 2022 before declining to approximately 1,475 days in 2025, still representing a 21% increase over 2017 baseline levels. These findings collectively demonstrate that the exponential growth of reported vulnerabilities has outpaced the remediation capacity of maintainers.

To address these scaling challenges, we evaluate LLMs as automated gatekeepers, uncovering a complex landscape of performance and operational trade-offs. Our results show that models like `GPT-5.1` achieve the top overall performance (62.8% F_1 on vulnerability detection), whereas reasoning-focused models such as `DeepSeek-R1` incur high latency, and high-recall models like `LLaMA-2` produce excessive false positives. Furthermore, many models exhibit a prevalence bias by frequently hallucinating common tags like CWE-79 (XSS). These findings suggest that the future of OSS security cannot rely solely on manual review. Instead, we advocate for a hybrid approach of using LLMs to flag potential vulnerability introduction at the publication stage, while enforcing stricter limits on high-risk ecosystem features like installation hooks.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback. This project was partially supported by Microsoft AI Security, the U.S. National Science Foundation (NSF) under Grant Nos. 2219920 and 2453496, and the Intergovernmental Personnel Act Independent Research & Development Program. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

1. Github advisory database, https://github.com/github/advisory-database, [Online; accessed 3. Oct. 2024]
2. Open Source Vulnerability format - Open Source Vulnerability schema (Oct 2024), https://ossf.github.io/osv-schema/, [Online; accessed 3. Oct. 2024]
3. Academy, S.: Best languages for web development, https://www.scaler.com/blog/best-language-for-web-development/, accessed: 2024-12-16
4. Ahmed, M.B.U., Harzevili, N.S., Shin, J., Pham, H.V., Wang, S.: Secvuleval: Benchmarking llms for real-world c/c++ vulnerability detection. arXiv preprint arXiv:2505.19828 (2025)
5. Akhavani, S.A., Ousat, B., Kharraz, A.: Open source, open threats? investigating security challenges in open-source software (2025), https://arxiv.org/abs/2506.12995
6. Alfadel, M., Costa, D.E., Shihab, E.: Empirical analysis of security vulnerabilities in python packages. Empirical Software Engineering **28**(59) (2023). https://doi.org/10.1007/s10664-022-10278-4
7. Bagmar, A., Wedgwood, J., Levin, D., Purtilo, J.: I know what you imported last summer: A study of security threats in thepython ecosystem (2021), https://arxiv.org/abs/2102.06301
8. Birsan, A.: Dependency confusion: How i hacked into apple, microsoft and dozens of other companies. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610 (2021), accessed: 2025-05-15
9. Bleeping Computer: Hundreds of malicious python packages found stealing sensitive data (2023), https://www.bleepingcomputer.com/news/security/hundreds-of-malicious-python-packages-found-stealing-sensitive-data/, accessed: 2024-10-14
10. BrowserStack: Best languages for web development, https://www.browserstack.com/guide/best-language-for-web-development, accessed: 2024-12-16
11. Center of Internet Security: The SolarWinds Cyber-Attack: What You Need to Know. https://www.cisecurity.org/solarwinds (Accessed: 06-30-2023)
12. Cisco Talos: Active exploitation of the MOVEit Transfer vulnerability by Clop ransomware group. https://blog.talosintelligence.com/active-exploitation-of-moveit/ (Accessed: 06-30-2023)
13. Costa, A.N., Medeiros, F.L., Dantas, J.P., Geraldo, D., Soma, N.Y.: Formation control method based on artificial potential fields for aircraft flight simulation. Simulation **98**(7), 575–595 (2022)
14. CyberSaint: Impact of using open source software on cybersecurity (2023), https://www.cybersaint.io/blog/impact-of-using-open-source-software-on-cybersecurity, accessed: 2024-10-15
15. Cybersecurity, (CISA), I.S.A.: Cisa open source software security roadmap. https://www.cisa.gov/resources-tools/resources/cisa-open-source-software-security-roadmap (2023), accessed: 2025-01-22
16. Dantas, J.P., Silva, S.R., Gomes, V.C., Costa, A.N., Samersla, A.R., Geraldo, D., Maximo, M.R., Yoneyama, T.: Asapy: A python library for aerospace simulation analysis. In: Proceedings of the 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. pp. 15–24 (2024)
17. DataDog: Malicious software packages dataset. https://github.com/DataDog/malicious-software-packages-dataset/ (2025)

18. Decan, A., Mens, T., Claes, M.: An empirical comparison of dependency issues in oss packaging ecosystems. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 2–12. IEEE (2017)
19. Decan, A., Mens, T., Constantinou, E.: On the impact of security vulnerabilities in the npm package dependency network. In: Proceedings of the 15th International Conference on Mining Software Repositories. pp. 181–191 (2018)
20. Decan, A., Mens, T., Grosjean, P.: An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering **24**(1), 381–416 (2019)
21. Du, X., Zheng, G., Wang, K., et al.: Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. arXiv preprint (2024)
22. Duan, R., et al.: Towards measuring supply chain attacks on package managers for interpreted languages. arXiv preprint arXiv:2002.01139 (2021), presented at NDSS
23. Fan, J., Li, Y., Wang, S., Nguyen, T.N.: Ac/c++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th international conference on mining software repositories. pp. 508–512 (2020)
24. Ghiasi, M., Niknam, T., Wang, Z., Mehrandezh, M., Dehghani, M., Ghadimi, N.: A comprehensive review of cyber-attacks and defense mechanisms for improving security in smart grid energy systems: Past, present and future. Electric Power Systems Research **215**, 108975 (2023)
25. Guo, W., Xu, Z., Liu, C., Huang, C., Fang, Y., Liu, Y.: An empirical study of malicious code in pypi ecosystem. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 166–177 (2023). https://doi.org/10.1109/ASE56229.2023.00135
26. Harrison, Philip, B.D.: Introducing npm package provenance (Apr 2023), https://github.blog/security/supply-chain-security/introducing-npm-package-provenance/, accessed: 2024-10-14
27. Hunters, G.B.: The family of safe golang libraries is growing (2023), https://bughunters.google.com/blog/4925068200771584/the-family-of-safe-golang-libraries-is-growing, accessed: 2024-10-14
28. IBM: Log4shell: The vulnerability explained (2024), https://www.ibm.com/topics/log4shell, accessed: 2024-10-14
29. Internet Archive: Wayback machine (2024), http://web.archive.org/
30. Jaffal, N.O., Alkhanafseh, M., Mohaisen, D.: Large language models in cybersecurity: A survey of applications, vulnerabilities, and defense techniques. AI **6**(9), 216 (2025)
31. Kikas, R., et al.: Structure and evolution of package dependency networks. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). pp. 102–112. IEEE (2017)
32. Kim, T., Park, S., Kim, H.: Why johnny can't use secure docker images: Investigating the usability challenges in using docker image vulnerability scanners through heuristic evaluation. In: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses. pp. 669–685 (2023)
33. Kluban, M., Mannan, M., Youssef, A.: On measuring vulnerable javascript functions in the wild. In: Proceedings of the 2022 ACM on Asia conference on computer and communications security. pp. 917–930 (2022)
34. Kurmi, A.: axios compromised on npm - malicious versions drop remote access trojan. StepSecurity (March 2026), https://www.stepsecurity.io/blog/axios-compromised-on-npm-malicious-versions-drop-remote-access-trojan, accessed: 2026-04-14

35. Lawrence Abrams: US govt offers $10 million bounty for info on Clop ransomware. https://www.bleepingcomputer.com/news/security/us-govt-offers-10-million-bounty-for-info-on-clop-ransomware/ (Accessed: 06-30-2023)
36. Lekssays, A., Mouhcine, H., Tran, K., Yu, T., Khalil, I.: {LLMxCPG}:{Context-Aware} vulnerability detection through code property {Graph-Guided} large language models. In: 34th USENIX Security Symposium (USENIX Security 25). pp. 489–507 (2025)
37. Libraries.io: Discover open source libraries. https://libraries.io/ (2024)
38. Lin, J., Mohaisen, D.: From large to mammoth: A comparative evaluation of large language models in vulnerability detection. In: Network and Distributed System Security (NDSS) Symposium 2025 (2025), https://www.ndss-symposium.org/wp-content/uploads/2025-1491-paper.pdf
39. MITRE: Mitre corporation. https://www.mitre.org/, accessed: 2024-10-13
40. MITRE: Common vulnerabilities and exposures (cve). https://cve.mitre.org/about/ (2024), accessed: 2024-10-13
41. MITRE: Cwe-699: Software development view. https://cwe.mitre.org/data/definitions/699.html (2025), accessed: 2026-04-16
42. Noever, D.: Can large language models find and fix vulnerable software? arXiv preprint arXiv:2308.10345 (2023)
43. npm: coa - package, https://www.npmjs.com/package/coa, accessed: 2024-10-14
44. npm: rc - npm package, https://www.npmjs.com/package/rc, accessed: 2024-10-14
45. Ohm, M., Plate, H., Sykosch, A., Meier, M.: Backstabber's knife collection: A review of open source software supply chain attacks. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA, Lisbon, Portugal, June 24–26, 2020, Proceedings 17. pp. 23–43. Springer (2020)
46. Pashchenko, I., et al.: Vulnerable open source dependencies: Counting those that matter. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (2018)
47. Phylum: Dozens of npm packages caught attempting to deploy reverse shell (2023), https://blog.phylum.io/dozens-of-npm-packages-caught-attempting-to-deploy-reverse-shell/
48. Snyk: Snyk vulnerability database. https://snyk.io/vuln, accessed: 2024-10-13
49. Snyk: npm dependency confusion attack targeting gxm reference. https://snyk.io/blog/npm-dependency-confusion-attack-gxm-reference/ (2022), accessed: 2025-05-15
50. Snyk: SNYK-JS-NODEIPC-2426370: Malicious Package in node-ipc. https://security.snyk.io/vuln/SNYK-JS-NODEIPC-2426370 (2022), accessed: 2025-05-16
51. Snyk: SNYK-JS-SWIGGYPRIVATEANALYTICS-8553970: Malicious Package in @swiggy-private/analytics. https://security.snyk.io/vuln/SNYK-JS-SWIGGYPRIVATEANALYTICS-8553970 (2024), accessed: 2025-05-25
52. of Standards, N.I., (NIST), T.: National vulnerability database (nvd). https://nvd.nist.gov/, accessed: 2024-10-13
53. Sun, Y., Wu, D., Xue, Y., et al.: Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. arXiv preprint (2024)
54. Sushma, D., Nalini, M., Kumar, R.A., Nidugala, M.: To detect and mitigate the risk in continuous integration and continues deployments (ci/cd) pipelines in supply chain using snyk tool. In: 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS). pp. 1–10. IEEE (2023)
55. Sysdig: The hidden economy of open source software (2023), https://sysdig.com/blog/hidden-economy-of-open-source-software/, accessed: 2024-10-15

56. Team, C.: Common weakness enumeration (cwe). https://cwe.mitre.org/ (2024), accessed: 2024-10-13
57. Team, G.: Go modules reference. https://go.dev/ref/mod (2023), online; Accessed: 2024-10-12
58. Team, G.: Go proxy. https://proxy.golang.org/ (2023), online; Accessed: 2024-10-12
59. The Hacker News: 27 malicious pypi packages with info-stealing capabilities discovered (2023), https://thehackernews.com/2023/11/27-malicious-pypi-packages-with.html, accessed: 2024-10-14
60. The White House: Summary of the 2023 request for information on open source software security. https://www.whitehouse.gov/wp-content/uploads/2024/08/Summary-of-the-2023-Request-for-Information-on-Open-Source-Software-Security.pdf (2024), accessed: 2024-10-07
61. Tyler, J.M., Murch, B.J., Vasilakis, C., Wood, R.M.: Improving uptake of simulation in healthcare: User-driven development of an open-source tool for modelling patient flow. Journal of Simulation **17**(6), 765–782 (2023)
62. United States Congress: Securing open source software act. https://www.govinfo.gov/content/pkg/BILLS-117s4913is/pdf/BILLS-117s4913is.pdf (2021), accessed: 2024-10-07
63. Valais, M.: Using the GO111MODULE everywhere! https://maelvls.dev/go111module-everywhere/#go111module-with-go-116, online; Accessed: 2024-10-12
64. vUpgradeU: Open source development: A double-edged sword (2023), https://www.vupgradeu.com/post/open-source-development-a-double-edged-sword, accessed: 2024-10-15
65. Wright, N.L., Nagle, F., Greenstein, S.: Open source software and global entrepreneurship. Research Policy **52**(9), 104846 (2023)
66. Yang, H., Liu, X.Y., Wang, C.D.: Fingpt: Open-source financial large language models. arXiv preprint arXiv:2306.06031 (2023)
67. Yildiz, A., Teo, S.G., Lou, Y., Feng, Y., Wang, C., Divakaran, D.M.: Benchmarking llms and llm-based agents in practical vulnerability detection for code repositories. In: Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 30848–30865 (2025)
68. Zahan, N., Zimmermann, T., Godefroid, P., Murphy, B., Maddila, C., Williams, L.: What are weak links in the npm supply chain? In: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. pp. 331–340 (2022)
69. Zahan, N., et al.: Do software security practices yield fewer vulnerabilities? In: 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 231–240. IEEE (2023)
70. Zerouali, A., et al.: On the impact of security vulnerabilities in the npm and rubygems dependency networks. Empirical Software Engineering **27**(5), 107 (2022)
71. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Advances in neural information processing systems **32** (2019)
72. Zhu, H., Li, J., Gao, C., et al.: Specification-guided vulnerability detection with large language models. arXiv preprint (2025)
73. Zimmermann, M., et al.: Small world with high risks: A study of security threats in the npm ecosystem. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 995–1010 (2019)

A Appendix: Reported Vulnerability Trends

A.1 Discouraged and Prohibited CWEs

MITRE has flagged a set of CWEs as too broad or vague for actionable mitigation and discouraged their use. We found 6,363 vulnerability reports associated with these CWEs and removed them from our dataset to get more accurate insights. The list of CWEs considered discouraged or prohibited in our analysis is derived programmatically by crawling the CWE page and checking the vulnerability mapping status for each entry, ensuring consistency with the official MITRE definitions. Examples of such cases are CWE-400 (Uncontrolled Resource Consumption) and CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor). MITRE [41] provides a structured view of software weaknesses that organizes high-level classes as prohibited.

A.2 Top CWE Descriptions

Table 5. Common Vulnerability Patterns and Ecosystem-Specific CWEs Across Software Platforms [56].

CWE	Ecosystem(s)	Description
CWE-79 (Cross-Site Scripting)	Packagist, Maven NPM, PyPI	Major concern in web applications, dominating Packagist, Maven, NPM, and PyPI.
CWE-22 (Path Traversal)	NPM, Maven PyPI	Enables access to files outside limited boundaries; highest in NPM, Maven, and PyPI.
CWE-94 (Code Injection)	NuGet, Packagist NPM, Maven, PyPI	High risk in dynamic server-side applications that handle user input; most common in NuGet and Packagist.
CWE-1321 (Prototype Pollution)	NPM	Highlights the need for validation mechanisms in object-handling libraries.
CWE-122 (Heap Buffer Overflow)	C/C++	Top-reported CWE for C/C++, reflecting the challenges of manual memory management.
CWE-416 (Use After Free)	C/C++, Crates	Leads to undefined behavior or security vulnerabilities; mostly observed in C and Crates.
CWE-787 (Out-of-Bounds Write)	NuGet	Decreasing in NuGet, likely due to memory safety features such as C# 8's Index/Range types.
CWE-362 (Race Condition)	C/C++, Crates	Historically prominent in C and Crates, but becoming less frequent as concurrency handling improves.

A.3 Total Packages in Each Package Manager

Figure 6 shows the overall increasing trend for total number of packages in different ecosystems. There are exceptions where certain platforms show a temporary decline in package count during specific years, which are likely attributed to the removal of outdated and unsupported packages from their registry. For instance, Golang experienced a notable package drop in 2021 following the release

of Golang 1.16, which introduced major changes to its module count and dependency management system [57,58,63]. In particular, before Go Modules, Go used a global workspace, defined by the GOPATH environment variable, to manage dependencies where all projects shared the same workspace, making it difficult to manage multiple versions of the same dependency. This led to version conflicts, dependency issues, and challenges in ensuring consistent builds across projects.

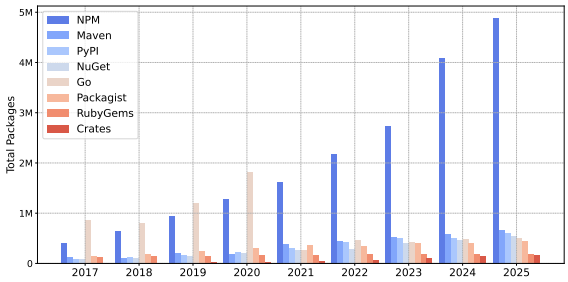

Fig. 6. Overview of total packages existing in each package manager. Extracted from libraries.io.

A.4 Terminology Definitions.

Below, we provide the formulas and definitions for each term used in our study to ensure clarity and precision.

Vulnerability Lifespan. The vulnerability lifespan measures the duration a vulnerability persists in a package before being patched. Since a single package may have multiple affected version ranges, we calculate the total lifespan by summing the durations of all such ranges. This provides a comprehensive view of how long vulnerabilities remain unresolved across different versions.

$$\sum_{i=1}^{n} (\text{First Patched Version Release Date}_i - \text{First Affected Version Release Date}_i)$$

Here, n represents the number of affected version ranges for a given package.

Time-to-Fix. Time-to-Fix measures the duration between the disclosure of a vulnerability and the release of a patched version of the affected package. It is calculated by subtracting the disclosure date of a report from the release date of the first unaffected version. This metric highlights how quickly ecosystems respond to disclosed vulnerabilities.

Vulnerability Disclosure. Common Vulnerabilities and Exposures (CVE) [40], managed by MITRE [39], is a reference list of publicly known vulnerabilities, each assigned a unique identifier. Complementing CVE, Common Weakness Enumeration (CWE) [56] serves as a community-developed taxonomy for identifying software weaknesses. Each weakness in the CWE list is assigned a unique identifier, simplifying the tracking and addressing of these issues across platforms.

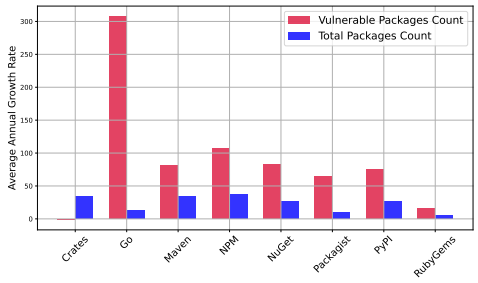

Fig. 7. Average annual growth rate of vulnerable packages and total packages by platform. Vulnerabilities have grown at a much faster rate than total packages in nearly all ecosystems.

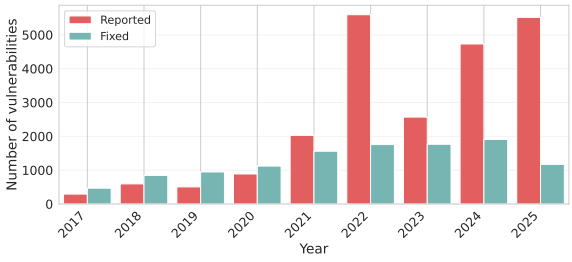

Fig. 8. Comparison of Reported Vulnerabilities against the number of Deployed Fixes.

Lastly, the National Vulnerability Database (NVD) [52] enhances CVE data with additional information, improving the context and understanding of vulnerabilities.

A.5 Vulnerability Reports vs Fixes vs Total Packages

Figure 7 illustrates the comparative growth rates of vulnerabilities and packages across all measured ecosystems over the years. An exception to the trend is the Rust ecosystem (Crates), which shows a slight negative growth rate in vulnerable packages (-1.26%) despite substantial growth in total packages (34.38%). This divergence indicates that, unlike other ecosystems, Crates has not experienced a comparable increase in reported vulnerabilities, potentially reflecting the impact of language-level safety features and stronger ecosystem governance.

B Appendix: Top CWEs Across Ecosystems

B.1 Top CWEs Responsible for Most CVEs

Figure 9 shows that six CWEs account for over 50% of the vulnerabilities across the target platforms. Figure 10 illustrates the top five CWEs over time.

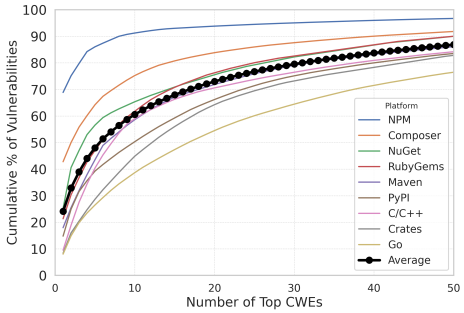

Fig. 9. Cumulative graph of vulnerabilities based on top CWEs.

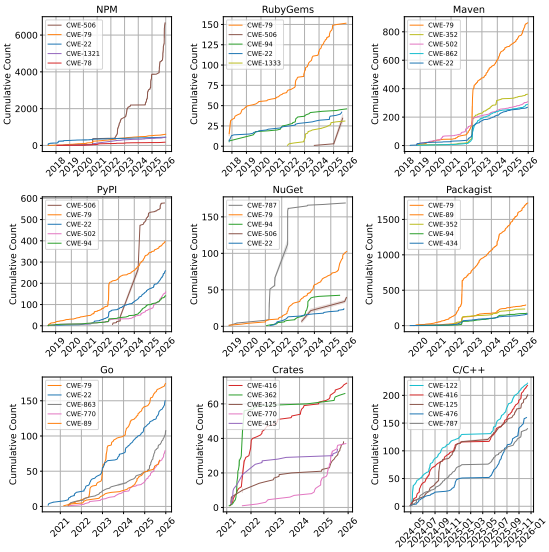

Fig. 10. Top five most reported CWEs in each ecosystem over time.

B.2 node-ipc Protestware

In March 2022, the maintainer of the popular `node-ipc` module in NPM, which has more than 600K weekly downloads, introduced geolocation-based file corruption logic in versions 10.1.1 and 10.1.2, targeting users in Russia and Belarus as a political statement against the invasion of Ukraine. The package contained malicious code that targeted users with IPs located in Russia or Belarus, and overwrote their files with a heart emoji [50]. This incident, later removed in version 10.1.3, marked one of the most prominent examples of "protestware". It illustrates that even trusted packages can become vectors for CWE-506 when maintainers deliberately inject malicious code. This issue was assigned CVE-2022-23812.

C Appendix: LLM Vulnerability Analysis and Prompts

Listing 1.1 shows the prompt used in the experiments to identify vulnerabilities. By changing the prompt to check the removed lines instead of the added ones, we create our negative samples for the classification.

C.1 Detecting Top CWEs with LLMs

```
You are a security code reviewer analyzing {language} code, provided with a diff file
      that shows the changes between the OLD and NEW code.
TASK: Determine if adding the NEW code (lines marked with "+") and removing the OLD code
      (lines marked with "-") would introduce ANY of these vulnerabilities:
{cwe_details}
In other words, is the NEW code vulnerable to any of the listed CWEs?

DIFF TO ANALYZE:
{diff_content}

Respond with ONLY this JSON (no other text):
{"vulnerable": true/false, "cwe_found": "CWE-XXX" or null, "evidence": "brief description
       of the vulnerability in the NEW code, or null if safe"}
Set vulnerable=true if the NEW code (+ lines) introduces ANY vulnerability from the list.
```

Listing 1.1. Prompt template for diff-based vulnerability classification.

C.2 Detecting CWE-506 with LLMs

Table 6. LLM Detection Rates for CWE-506

Model	Detection Rate
CodeGemma_7B	6.7%
DeepSeek-Coder-V2_16B	46.5%
DeepSeek-R1_70B	76.8%
GPT-4o	80.4%
GPT-5.1	83.5%
LLaMA-2_70B	70.4%
LLaMA-3.3	83.3%
Mixtral-8x7B	72.9%
Phi-3_14B	26.5%
Qwen-2.5-Coder_32B	74.4%

```
You are a security code reviewer analyzing {language} code. Analyze the following {self.config.target_language} source file for security vulnerabilities.

Target file: {filename}

ONLY evaluate for Common Weakness Enumerations CWE-506 Embedded Malicious Code. Do NOT report any other types of issues outside of the scope.

### Analysis Rules:
  - If a CWE cannot be conclusively identified, do NOT include it.
  - Every reported vulnerability MUST include:
  - Precise line numbers
  - The exact vulnerable code snippet
  - An explanation of why it matches the CWE definition

### Code to analyze:
```{self.config.target_language}
{code}
```
Output Format with no commentary outside JSON:

{{
"filename": "{filename}",
"vulnerabilities_found": [
    {{
    "cwe_id": "CWE-XXX",
    "title": "Short CWE name",
    "description": "Clear explanation tied to the code",
    "code_snippet": "Relevant code snippet showing the vulnerability",
    "location": "Function information of the vulnerability location "
    }}
],
"overall_assessment": "2 sentences summary if vulnerabilities are found, otherwise set it to 'No vulnerabilities found'",
"safe": true|false
}}

If NO vulnerabilities are found for the specified CWEs:
- Return an empty vulnerabilities_found array
- Set safe to true
- Set overall_assessment to "No vulnerabilities found"

Ensure the JSON is valid and machine-parsable.
```

Listing 1.2. Prompt template for CWE-506 vulnerability classification.