Detecting Protracted Vulnerabilities in Open Source Projects

ARJUN SRIDHARKUMAR*, University of Toronto, Canada
SARA AL HAJJ IBRAHIM*, University of Toronto, Canada
JIAYUAN ZHOU, Queen's University, Canada
YULIANG WANG, University of Toronto, Canada
SAFWAT HASSAN, University of Toronto, Canada
AHMED E. HASSAN, Queen's University, Canada
SHURUI ZHOU, University of Toronto, Canada

Timely resolution and disclosure of vulnerabilities are essential for maintaining the security of open-source software. However, many vulnerabilities remain unreported, unpatched, or undisclosed for extended periods, exposing users to prolonged security threats. While various vulnerability detection tools exist, they primarily focus on predicting or identifying known vulnerabilities, often failing to capture vulnerabilities that experience significant delays in resolution. In this study, we examine the vulnerability lifecycle by analyzing protracted vulnerabilities (PCVEs), which remain unresolved or undisclosed over long periods. We construct a dataset of PCVEs and conduct a qualitative analysis to uncover underlying causes of delay. To assess current automated solutions, we evaluate four state-of-the-art (SOTA) vulnerability detectors on our dataset. These tools detect only 1,059 out of 2,402 PCVEs, achieving approximately 44% coverage. To address this limitation, we propose *DeeptraVul*, an enhanced detection approach designed specifically for protracted cases. *DeeptraVul* integrates multiple development artifacts and code signals, supported by a Large Language Model (LLM)-based summarization component. For comparison, we also evaluate a standalone LLM. Our results show that *DeeptraVul* improves detection performance, achieving a 14% increase in coverage across all PCVEs and reaching 90% coverage on the *DeeptraVul* PCVE subset, outperforming existing SOTA detectors and standalone LLM based inference.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Information systems → Language models**.

Additional Key Words and Phrases: Open Source Software, Empirical Study, AI for Software Engineering, Software Security, Vulnerability Detection

1 Introduction

Practitioners and researchers have developed protocols for the process of addressing vulnerabilities. Among these, the Coordinated Vulnerability Disclosure (CVD), or responsive disclosure model [1, 25], outlines how vulnerability reporters and open-source software (OSS) developers should coordinate to ensure that vulnerability details are not disclosed until the issues are resolved [29, 34, 36, 37, 41, 55]. As part of this process, OSS maintainers are also

*These authors contributed equally to this work.

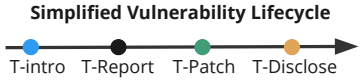
Authors' addresses: Arjun Sridharkumar, University of Toronto, Canada, arjunsridharkumar@mail.utoronto.ca; Sara Al Hajj Ibrahim, University of Toronto, Canada, sara.alhajjibrahim@mail.utoronto.ca; Jiayuan Zhou, Queen's University, Canada, jiayuan.zhou@queensu.ca; Yuliang Wang, University of Toronto, Canada, stevenyuliang.wang@mail.utoronto.ca; Safwat Hassan, University of Toronto, Canada, safwat.hassan@utoronto.ca; Ahmed E. Hassan, Queen's University, Canada, ahmed@cs.queensu.ca; Shurui Zhou, University of Toronto, Canada, shurui.zhou@utoronto.ca.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1557-7392/2026/4-ART
https://doi.org/10.1145/3809490

Just Accepted

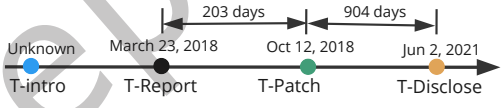
advised to disclose the vulnerability as soon as a fix becomes available, since users depend on this information to stay informed and apply necessary updates [42, 47, 81].



In the figure on the right, we illustrate a simplified vulnerability lifecycle comprising four key events: (1) $\mathbf{T_{Intro}}$, when a vulnerability is introduced into the codebase; (2) $\mathbf{T_{Report}}$, when the vulnerability is reported to the OSS developers; (3) $\mathbf{T_{Patch}}$, when the fix for the vulnerability becomes available; and (4) $\mathbf{T_{Disclose}}$, when security advisories make vulnerability information available to the public. According to the previously described best practices, it is crucial to minimize the time difference between each pair of the events, such as $T_{Patch} - T_{Report}$ and $T_{Disclose} - T_{Patch}$.

However, these best practices are often neglected. Developers sometimes fix vulnerabilities silently without intending to publicly disclose them [81]. Furthermore, previous studies have highlighted delays at various stages of the vulnerability lifecycle. For example, Decan et al. examined 369 vulnerability reports affecting 269 NPM packages and found a median delay of 11 days for vulnerabilities to be fixed after being reported [23]. Additionally, certain vulnerabilities were only resolved following their public disclosure. In a study by Alfadel et al. [9], 1,396 vulnerability reports concerning 686 Python packages were analyzed, revealing that 40.86% of these vulnerabilities were fixed post-public disclosure, with a median delay of two months from the initial reporting of the vulnerability. Prior works have also found that the interval between patching and public disclosure can vary widely, ranging from days to years [45, 47, 81].

In the figure on the right, we illustrate the timeline of *CVE-2021-3538*, as analyzed in our study, showing delays of 203 days in T_{Patch} and 904 days in $T_{Disclose}$. These delays expose vulnerabilities that malicious third parties can exploit, posing risks to end users who depend on public disclosure to integrate patches [9].



Timeline of CVE-2021-3538

Although previous research has found that delays throughout the vulnerability timeline are common [68], the underlying causes of the large delays and potential solutions to mitigate the associated risks remain unclear. Consequently, in this project, we conducted a comprehensive study to understand the factors that contribute to delays, particularly in patching (T_{Patch}) and public disclosure ($T_{Disclose}$). To enhance clarity in this paper, we define vulnerabilities with significant gaps between any of the timestamps (i.e., $T_{Report} - T_{Patch}$ or $T_{Patch} - T_{Disclose}$) as **protracted vulnerabilities** (***PCVEs***).

In this work, we study whether a vulnerability is present during the prolonged period before it is recognized or disclosed. In such protracted cases, evidence of an underlying vulnerability may surface incrementally across different development artifacts and at different points in time, rather than appearing only in finalized source code or as a completed security patch. In practice, vulnerability detection is most commonly framed as a source-code analysis problem and generally excludes information beyond the code itself [50]. Our setting instead asks whether a vulnerability is already present based on the evidence available at a given point in time, regardless of which development artifacts provide that evidence.

Consequently, SOTA techniques are applied to the artifacts available at each point in time in the same technical manner as originally designed, but with a different evaluation objective. Rather than assuming the presence of finalized code or completed fixes, these techniques are used to assess whether intermediate artifacts exhibit signals consistent with an underlying vulnerability. This usage reflects how protracted vulnerabilities unfold in practice, where relevant signals surface incrementally across artifacts prior to formal recognition. Incorporating multiple artifact types therefore enables analysis of the extended pre-disclosure period during which the vulnerability exists but has not yet been identified as security relevant.

Just Accepted

To the best of our knowledge, there is no publicly available dataset consisting of PCVEs and corresponding artifacts, including code changes and vulnerability-related discussions. Thus, we first parsed the National Vulnerability Database (NVD) [5], one of the popular security repositories, to collect 20,951 CVEs that have development artifacts on GitHub, spanning the period from 1999 through July 2024; then we gathered a subset of CVEs with potential delays of more than one year, resulting in a total of 3,228 CVEs (15.4%) classified as PCVEs.

To achieve our research objectives, we formulate three research questions outlined below. Figure 1 provides an overview of our project along with the associated research methods.

RQ1: What factors contribute to delays throughout the lifecycles of the PCVEs? To understand the potential causes of delays, we conducted a mixed-methods analysis on the lifecycle of 94 stratified sampled PCVEs. This stratified sample was chosen to ensure a 95% confidence level with a 10% margin of error, representing a portion of the total CVEs in the dataset. The analysis reveals eight key factors contributing to delays in PCVE resolution, with the most prevalent being (a) delayed reporting to the NVD, (b) misjudgment of vulnerability severity and relevance, and (c) lack of active project maintenance. Among the 94 CVEs analyzed, the median time to patch vulnerabilities was 9 days, while the median time to disclose vulnerabilities publicly was 619 days.

RQ2: To what extent do the SOTA vulnerability detection methods demonstrate efficacy in identifying PCVEs? Based on the results from RQ1 concerning the complexities of addressing vulnerabilities, it is evident that previous SOTA vulnerability detection approaches have been limited to a narrow range of artifacts. Consequently, our objective is to evaluate the effectiveness of four SOTA methods (i.e., *LineVul* [31], *DeepDFA* [74], *VulCurator* [57], *MemVul* [62], and *PatchRNN* [77]).

The evaluation results show that out of 2,402 PCVEs, 1,059 (44%) were detected by the selected SOTA methods. Also, each SOTA method is good at identifying a unique group of PCVEs, which raises the question naturally of if we could take advantage of the collective effort to detect more PCVEs. Subsequently, we conducted a qualitative analysis on a subset of the remaining 1,343 PCVEs, which represent 56% of the dataset. This analysis yielded five critical insights aimed at enhancing PCVE detection. These insights encompass potentially valuable textual and non-textual artifacts that could contribute to the creation of a more effective vulnerability detection method.

RQ3: How can the SOTA be improved to identify PCVEs that currently remain undetected? Building on the findings from RQ1 and RQ2, that current SOTA performs poorly due to input artifact restrictions and can be improved by including external knowledge and additional artifacts, we performed a feasibility study to understand the potential of leveraging various input artifacts, including issues, commits, PRs, and Common Weakness Enumeration (CWE) information, for the accurate identification of PCVEs. We designed *DeeptraVul*, a novel approach that integrates a comprehensive set of textual artifacts to identify additional PCVEs that existing SOTA methods fail to detect. For context, our implementation incorporates an LLM-based summarization

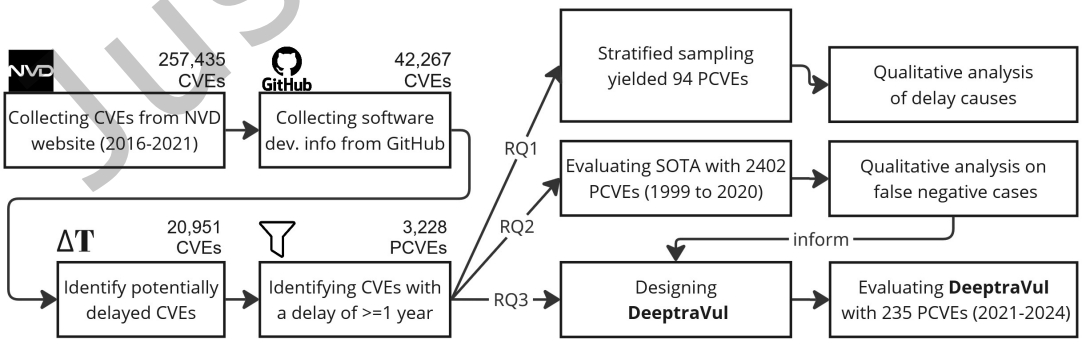


Fig. 1. Research method overview.

Just Accepted

component and includes a direct LLM baseline for comparison, allowing us to evaluate how LLMs behave relative to our model. Our approach achieves superior performance, detecting 14% more PCVEs overall and 46% more on the *DeeptraVul* dataset compared to existing SOTA methods. In addition, we conducted an analysis of the false negative instances in which *DeeptraVul* did not succeed in detection, uncovering additional indicators that highlight potential directions for future research.

This study emphasizes the widespread nature of protracted vulnerabilities, underscores the necessities for continued systematic investigation, and calls for the development of tooling support to accurately identify PCVEs and reduce delays in future research endeavors. Specifically, we make the following contributions:

- To the best of our knowledge, we are the first to study the causes of the delay in patching and disclosure of CVEs;
- We evaluate the SOTA vulnerability detection methods using the PCVE dataset;
- We conduct a feasibility study using *DeeptraVul*, an innovative method for detecting vulnerabilities leveraging a wide range of development artifacts and external vulnerability-related knowledge;
- We demonstrate that *DeeptraVul* surpasses current SOTA methods, including an LLM-based approach, in identifying PCVEs.
- We provide a comprehensive dataset of 3,228 PCVEs and a replication package [7] to support future research efforts.

2 Related Work

2.1 Vulnerability Timeline and Relevant Activities

Previous studies have explored various stages of the vulnerability lifecycle, each focusing on different timestamps and specific objectives [8, 9, 12, 23, 40, 42, 47, 56, 70, 73]. Figure 2 depicts the specific time frames examined in these prior studies. For instance, Nappa et al. investigated the patch deployment process for 1,593 vulnerabilities, finding that the time between patch releases ranged up to 118 days, with a median duration of 11 days [56]. Similarly, Bilge et al. assessed the prevalence and duration of zero-day attacks by examining the interval between exploit availability and public disclosure, using multiple vulnerability databases to automatically identify instances of zero-day attacks [12]. Li et al. conducted a large-scale analysis of 4,080 fixes related to 3,094 vulnerabilities, discovering that over a quarter (26.4%) of these security issues remained unpatched 30 days after disclosure [47].

In this project, we focus on the period between the discovery of a vulnerability (T_{Report}) and its public disclosure ($T_{Disclose}$). Our goal is to understand the reasons for delay and to develop heuristics for the early detection of vulnerabilities. Several related studies have also examined this time frame, though with varying research goals. For example, Decan et al. found that it took an average of 11 days to fix vulnerabilities reported in 369 instances affecting 269 NPM packages [22]. Similarly, Alfadel et al. analyzed 1,396 vulnerability reports in the PyPi ecosystem affecting 686 Python packages, and found that 40.86% of these vulnerabilities were addressed after public disclosure, with a median delay of two months. Other studies have investigated the use of social media platforms (e.g., Twitter, Reddit) for disseminating information about vulnerabilities throughout their lifecycle [40, 70]. Unlike previous studies, our research examines delays in the vulnerability lifecycle across a diverse set of open-source projects. Our goal is to identify the underlying causes of these delays and to develop more efficient methods for early vulnerability detection

2.2 SOTA Vulnerability Detection Methods

Researchers have designed several deep learning (DL)-based detectors to identify vulnerabilities, including function-level DL-based detectors [82], slice-based detectors [51, 52], and line-level detectors [38]. These approaches differ in their code representation granularity and contextual modeling strategies. To systematically assess their effectiveness, Steenhoek et al. comprehensively reviewed and benchmarked the performance of

Just Accepted

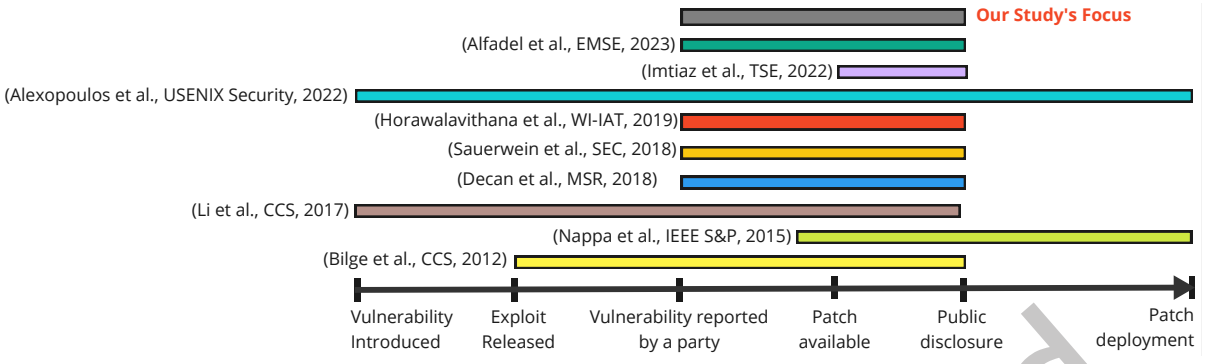


Fig. 2. Summary of the time frames within the vulnerability lifecycle examined by previous studies.

nine SOTA DL-based detectors on C/C++ projects [75]. Their findings revealed that *LineVul*, a transformer-based line-level vulnerability prediction method, exhibited the best overall performance among the evaluated models [31].

Building upon the success of transformer-based token models, subsequent research has explored enriching such models with additional program semantics. For example, *DeepDFA* incorporates control-flow and data-flow graphs (CFGs and DFGs) to improve detection accuracy on several benchmark datasets [74]. Empirical results show that *DeepDFA* outperforms prior token-based baselines by leveraging these additional structural representations. Consistent with prior work comparing vulnerability detectors [21, 49, 71], we therefore include both *LineVul* and *DeepDFA* as baselines for RQ2 and RQ3.

In addition, other studies have developed methods to detect vulnerabilities using artifacts beyond source code. For example, early work classified commit messages and corresponding code changes as either vulnerable or not using linear SVM models with Bag-of-Words representations [69]. This direction was later extended by *PatchRNN*, which uses recurrent neural networks to jointly model code diffs and commit messages for identifying security-relevant patches [77]. More recent approaches incorporate additional artifact information. *VulCurator* replaces traditional SVM techniques with transformer-based models and integrates issue reports, demonstrating improved detection performance [57]. Likewise, *MemVul* augments issue-report analysis with external CWE knowledge to improve classification accuracy [62]. Given that our CVE dataset includes both source code and development-related artifacts, we selected *PatchRNN*, *VulCurator*, and *MemVul* as additional baselines and evaluated their performance on our PCVE dataset.

3 RQ1: What factors contribute to delays throughout the lifecycles of the PCVEs?

This section begins by describing the process used to construct the PCVE dataset, followed by a detailed quantitative and qualitative analysis that highlights the delays observed throughout the vulnerability lifecycle.

3.1 Constructing the PCVE Dataset

(1) **Collecting CVEs disclosed from 1999 to 2024.** We use the National Vulnerability Database (NVD) as our primary data source and retrieve CVEs through the NVD Data Feeds APIs [6]. For each CVE, we collect its identifier, disclosure date, and references to external sources containing additional vulnerability information. This step resulted in a total of 257,435 CVEs.

(2) **Collecting development information from GitHub.** To gather detailed information on the lifecycle of vulnerabilities, including discussions and code changes, we focus our study on projects hosted on GitHub.

Just Accepted

This selection criterion requires at least one GitHub URL in the reference list, resulting in 42,267 CVEs. For each GitHub URL associated with commits, issues, or PRs, we use the GitHub API [26] to gather the relevant information.

(3) **Identifying CVEs that have potential delays during their lifecycle.** To study these delays, we compared the timestamps of collected GitHub artifacts with the NVD disclosure dates, retaining only those CVEs where earlier artifacts indicated a delay. This process resulted in the identification of 20,951 CVEs.

(4) **Sampling PCVEs with large time intervals between the artifacts creation and NVD disclosure.** We calculated the time difference (ΔT) between the earliest collected development artifact timestamp ($T_{Earliest}$) and the NVD disclosure date ($T_{Disclose}$) for each CVE. The log-scaled distribution of ΔT for 20,951 CVEs is shown in Figure 3b, with a summary in Figure 3a. This study focuses on CVEs with a ΔT of at least 365 days, identifying 3,228 CVEs (15.4%) as *PCVEs*.

The 365-day threshold is motivated by both the empirical distribution of ΔT and the conceptual notion of protracted vulnerabilities. Empirically, ΔT follows a highly skewed distribution (Figure 3b). The 75th percentile is 150 days and the mean is 218.6 days, while 365 days lies in the long-tail region that contains only 15.4% of CVEs. This indicates that CVEs with $\Delta T \geq 365$ days represent atypical cases rather than typical disclosure delays.

From a conceptual perspective, a one-year interval corresponds to a full development cycle for many software projects [46]. A vulnerability with an earliest development artifact dated more than one year prior to public disclosure remains present across multiple cycles of development, testing, and release. This extended duration reflects the defining characteristic of PCVEs. Accordingly, the 365-day threshold serves as the criterion for identifying protracted vulnerabilities in this study.

3.2 Distribution of the Delays during PCVEs' lifecycle

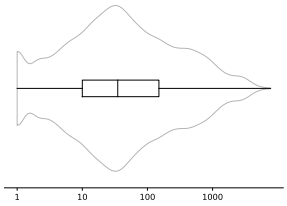
First, we aim to quantitatively analyze the delays occurring throughout the lifecycle of the PCVEs.

Stratified sampling. Manually reviewing each CVE's reference links, including understanding the vulnerability, its technical details, and related discussions, requires approximately three hours of focused effort per CVE. To keep the qualitative analysis manageable while preserving representativeness, we apply stratified sampling to the set of 3,228 CVEs [64]. The PCVEs are grouped into nine stratified buckets based on their ΔT values, using three-month intervals. From these buckets, we select a total of 94 samples, achieving a 95 percent confidence level with a 10 percent margin of error for the ΔT distribution.

Various types of vulnerability lifecycles. As previously discussed, we identify artifacts corresponding to each timestamp (i.e., T_{Report}, T_{Patch}, $T_{Disclose}$) for each PCVE and compute the intervals between them to examine delays. It is important to note that not all three timestamps are available for every PCVE, which leads to incomplete data in some cases. Among the 94 PCVEs selected, we successfully retrieved all three timestamps for 48 cases. Table 1 presents a categorization of the 94 PCVEs based on the presence and sequence of the three activities. We organize them into three distinct lifecycle types.

Statistic	# Days
Mean	218.6
25th Quartile	10
Median	35
75th Quartile	150
90th Quartile	597
95th Quartile	1,061

(a) Statistics of ΔT.



(b) Log-scaled distribution.

Fig. 3. Distribution of ΔT.

Just Accepted

(1) **PCVEs with only two recovered timestamps.**
[—●—●→] For 21 PCVEs, we could only obtain the T_{Patch}, with the T_{Report} remaining unknown, such as in the cases of *CVE-2018-11232* and *CVE-2020-27601*. The mean interval between T_{Patch} and $T_{Disclose}$ is 962 days.
[●—●→] For 22 PCVEs, only the timestamp for T_{Report} was determined, with no traceable patch artifacts, as seen with *CVE-2020-19268* and *CVE-2016-11014*. On average, the time gap between T_{Report} and $T_{Disclose}$ spans 714 days.

(2) **Following the CVD Model but with Delays.**
[●—●—●→] 39 PCVEs adhered to a sequence of reporting, patching, and subsequent public disclosure. The average time to patch these CVEs was 108 days, with a median resolution period of 9 days. The process of public disclosure took an average of 634 days, with a median of 592 days. These results align with the research conducted by Decan et al. and Alfadel et al., who examined the timelines for vulnerability patching within the NPM and PyPI ecosystems respectively [9, 23]. *CVE-2021-23418* is an example where, despite early reporting, patch development and public disclosure were delayed.

(3) **PCVEs that Violate the CVD Model.**
[●—●—●→] 6 PCVEs had their vulnerabilities patched only after public disclosure. On average, it took 708 days to resolve these vulnerabilities post-reporting, with a median of 676 days. These PCVEs were disclosed approximately 125 days on average before a patch was issued, with a median of 63 days. Such delays compromise the security of the OSS ecosystem, as hackers could exploit the disclosed vulnerabilities to target

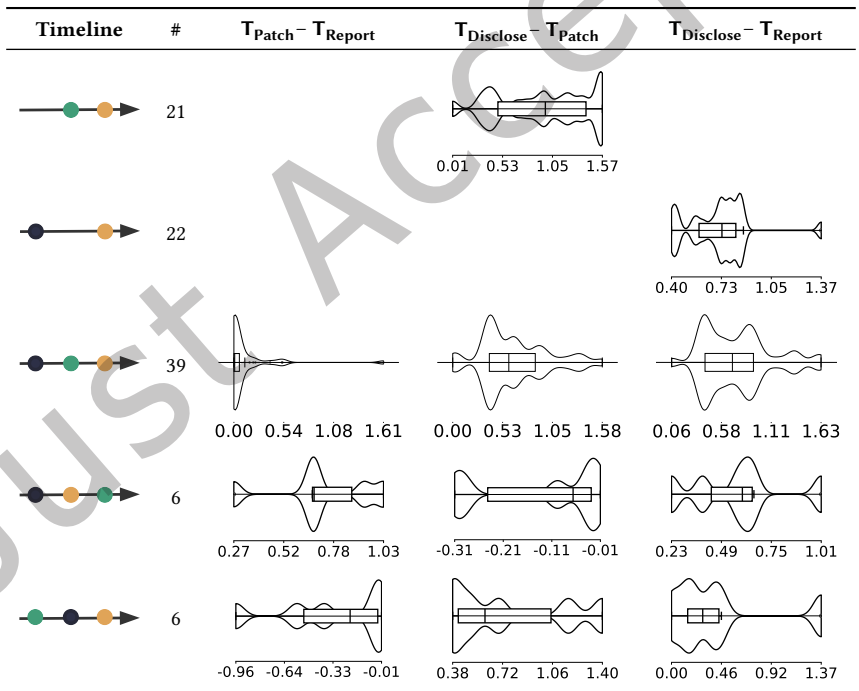


Table 1. Categorization of PCVE timeline types. The unit of measurement on the x-axis is thousands of days. ●– T_{Report}, ●– T_{Patch}, ●– $T_{Disclose}$.

Just Accepted

unaware end users. *CVE-2020-7731* was publicly disclosed before a patch was available, resulting in a delay between disclosure and resolution.

[●—●—▶] 6 PCVEs were reported only after patches had already been made available. On average, these vulnerabilities were patched 331 days prior to reporting, with a median interval of 214 days. Public disclosure of these PCVEs occurred, on average, 756 days after patching, with a median interval of 599 days. This pattern is indicative of silent fixes. Such practices among OSS developers result in delays in vulnerability disclosure, providing malicious third parties the opportunity to analyze code changes and exploit vulnerabilities. *CVE-2016-6786* involved delayed reporting and disclosure, resulting in limited timely visibility into the vulnerability.

3.3 Reasons for delays in PCVEs.

Next, we construct timelines for PCVEs to qualitatively examine the factors contributing to delays, followed by a summary of the root/observed causes underlying these delays.

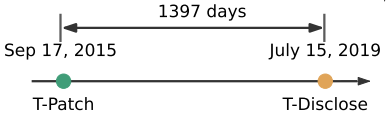
3.3.1 ***Qualitative Data Analysis.*** For each sampled PCVE, we reviewed discussion threads in referenced GitHub issues and PRs, if available, to determine potential reasons for the delays in either T_{Patch} and $T_{Disclose}$ respectively. Specifically, we used an open coding procedure [72, 78] to categorize these reasons. Note that a PCVE can have multiple causes throughout its lifecycle, contributing to delays in key activities such as reporting, fixing, and public disclosure. Thus, each PCVE might be assigned multiple codes.

One author conducted an initial analysis of 25 CVEs, developing a preliminary codebook to establish an initial categorization framework. To ensure consistency in labeling, two additional authors independently reviewed the coding decisions, engaging in collaborative discussions to refine the categorization process. Discrepancies were resolved through iterative dialogue, leading to the finalization of a consistent coding scheme. Following this initial phase, the team expanded the analysis to a stratified sample of 94 CVEs. This sample size was determined to achieve a 95% confidence level with a 10% margin of error, ensuring a representative subset of the dataset. As the analysis progressed, the emergence of new themes diminished, and by the time coding was completed for these 94 CVEs, no significant novel patterns were identified. To confirm this, an additional 25 CVEs were analyzed, contributing only marginal new insights and further supporting the conclusion that thematic saturation had been reached [30].

Throughout the study, an iterative and dynamic coding approach was used, with ongoing discussions to refine and validate the categorization framework. Any remaining discrepancies were resolved collaboratively, ensuring the robustness of the classification methodology.

3.3.2 ***Reason for delays in PCVEs lifecycle.*** Among the 94 sampled PCVEs, we identified eight potential factors contributing to the observed delays. Table 2 summarizes the codebook, providing a description of each factor with the distribution of PCVEs that exhibit delays in T_{Patch} or $T_{Disclose}$. We describe each identified factor in detail and present illustrative examples to support the analysis.

Category 1: Delayed Reporting to NVD. This category includes cases where a vulnerability was patched in a public codebase, but its corresponding entry in the NVD was published only after a significant delay. The delay we refer to is the time gap between the commit that fixed the issue (T_{Patch}) and the official disclosure date in the NVD ($T_{Disclose}$). We identified **65** such instances where vulnerabilities were patched in open-source projects, yet the corresponding NVD entries appeared days, weeks, months, or even years later. In reviewing these cases, we found no observable artifacts, such as GitHub issues, PRs, advisory discussions, or maintainer comments, addressing the delay between T_{Patch} and $T_{Disclose}$. For example, in *CVE-2019-1010308*, the GitHub repository contained a silent *commit#e1af89a* on September 17, 2015, addressing the vulnerability, while the disclosure happened on July 15, 2019. When the NVD eventually published the CVE, it referenced the



Just Accepted

Table 2. Codebook showcasing the underlying reasons behind the delays.

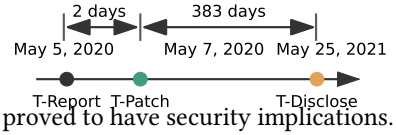
#	Reason for delay	Description	Incidence
1	**Delayed NVD Disclosure**	The vulnerability was fixed, but there was a delay in reporting it to the NVD or publishing the CVE, resulting in a gap between the fix and public disclosure. This may have been due to a silent fix or a slow reporting process.	65 (46.76%)
2	**Misjudgment of Relevance and Severity**	The issue was reported, but the maintainers either: 1. Explicitly stated it was a lower priority 2. Focused on other contributions during the same period 3. Debated whether it qualified as a vulnerability or reported it as a CVE.	27 (19.42%)
3	**Lack of Active Maintenance**	The issue has remained open, with the repository experiencing months of inactivity; lacking contributions, updates, or maintenance, resulting in problems staying unresolved for significant periods.	16 (11.51%)
4	**Incomplete or Insufficient Fix**	The initial fix did not fully address the issue, requiring further updates.	10 (7.19%)
5	**Disagreement on Resolution**	The team had conflicting views on how to resolve the issue.	8 (5.76%)
6	**Other or Unknown**	The delay occurred between the reported issue and the fix, but there is no clear evidence explaining the cause, making it difficult to determine what was delayed.	7 (5.04%)
7	**Resource Constraints**	The team lacked the necessary resources (e.g., time, developers, or testing facilities) to address the issue quickly.	4 (2.88%)
8	**Lack of Expertise**	The maintainers lacked the necessary expertise and explicitly sought external help.	2 (1.44%)

Just Accepted

commit directly but provided no further insights. This absence of intermediate documentation made it challenging to understand the reason for the delayed disclosure.

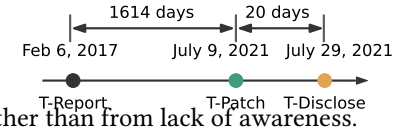
Category 2: Misjudgment of Relevance and Severity. This category includes cases where the vulnerability was reported, but maintainers either prioritized other contributions, considered the issue to be of lower severity, or questioned whether it warranted a CVE assignment, resulting in delayed action. These cases generally fall into three types.

Case 1: Issues are explicitly stated as lower priority. This type captures situations where maintainers clearly assigned the report a lower priority relative to other ongoing work. A representative example is *CVE-2020-10064*, reported in *zephyrproject-rtos/zephyr* on May 5, 2020. The issue was labeled as medium priority within the project's workflow, indicating that it was treated as a routine matter at the time. Maintainers closed the report on May 7, 2020, and added commit#38970c0 referencing the issue on the same day. The vulnerability was later assigned a CVE and publicly disclosed on May 25, 2021. This example illustrates cases where an issue appeared ordinary initially but later proved to have security implications.

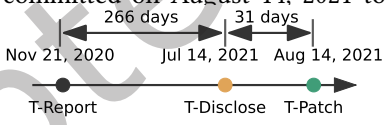


Case 2: Issues are deprioritized while maintainers focused on other contributions. This type describes reports that were acknowledged but repeatedly postponed, while maintainers focused on other development tasks. For this case, in *CVE-2021-23418*, the vulnerability was reported in the *nicolargo/glances* repository as *issue#1025*

on February 6, 2017. Although acknowledged and scheduled for Glances 2.9, it was repeatedly deferred across milestones—2.9, 2.9.1, 2.11, 3.0, 3.1.3, 3.1.7, and later. During this period, maintainers continued regular development activity,
including several commits shortly after the report (February 12 and February 19, 2017) that did not address the issue. A patch was applied on July 9, 2021, and public disclosure followed on July 29, 2021. This example illustrates how delays can stem from competing development priorities rather than from lack of awareness.



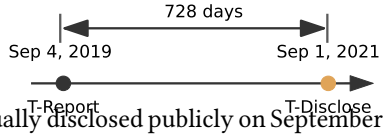
Case 3: Issues where maintainers were initially uncertain whether the report qualified as a vulnerability. This category captures situations in which the security relevance of a report was unclear and required additional assessment before being treated as a vulnerability. A representative example is *CVE-2020-36420* in the Gentoo ecosystem, documented in issue#755896, which was reported on November 21, 2020. In this case, the affected package was no longer maintained. During the discussion, the maintainers described it as a *"trivial issue with a minor impact"* and noted that they were *"not sure if there is a need to stabilize this package"*. Routine development continued while its potential security implications were evaluated. The vulnerability was publicly disclosed on July 14, 2021, and a patch was committed on August 14, 2021 to remove the unmaintained package. This case shows how delays may arise when the severity or security impact of a report is not immediately evident.



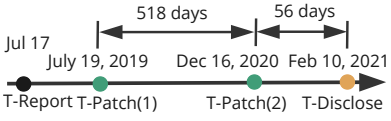
Overall, we identified **27** similar cases where delays resulted from misjudged severity or competing development priorities.

Category 3: Lack of Active Maintenance. This category includes vulnerabilities that remain unpatched due to the absence of sustained development activity. In many open-source projects, maintenance relies on voluntary contributions. When these contributions stop for a significant period, reported vulnerabilities may remain unresolved regardless of their severity. A clear example is the project *taosir/wtcms*, which has shown minimal activity since its creation. The commit history shows only a few short periods of development, including September 2017, April 2018, and June and December 2019.

For the remainder of the time between 2017 and 2025, the repository was inactive and no commits were made. On September 4, 2019, a vulnerability in this project was reported as *CVE-2020-20343*. No patch was provided, no discussion took place, and the issue remains unresolved at the time of writing. Although no corrective action was taken, the vulnerability was eventually disclosed publicly on September 1, 2021. This case shows how the lack of active maintenance can result in serious vulnerabilities remaining unpatched for extended periods. Of the 94 PCVEs analyzed in our study, **16** cases involved inactive projects.

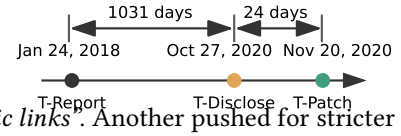


Category 4: Incomplete or Insufficient Fix. This category includes cases where open-source developers release patches that do not fully address the vulnerability, requiring multiple revisions. This delays both the release of a secure version and public disclosure, increasing user exposure. A clear example is a path traversal vulnerability in *QuorumDMS/ftp-srv*, which was later assigned *CVE-2020-26299*. It was first reported on July 17, 2019, in *issue#167*, with a *proof-of-concept video* shared in the discussion thread demonstrating the exploit. An initial fix was submitted two days later in *PR#168*. The patch was incomplete. One contributor wrote, *"I'd like to see if we can confirm this is still the case"*, and later testing confirmed that the issue was still present. On December 9, 2020, another contributor reported, *"I was able to reproduce it on 4cd88b1"*. A second fix was committed on December 16 via *commit#457b859*, but concerns remained. A user commented, *"Still not fixed or I am doing something wrong"*. The vulnerability was eventually disclosed publicly on February 10, 2021, after multiple incomplete fixes and continued concerns. This case shows how incomplete patches can delay remediation and increase the risk of exposing technical details that could be exploited. Of the 94 PCVEs analyzed, **10** cases fall into this category.



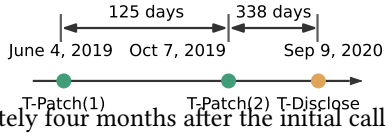
Just Accepted

Category 5: Disagreement on Resolution. This category includes cases where fixing the vulnerability was delayed due to disagreements among developers about how to handle the issue. These discussions often extended over time, slowing down decision-making and implementation. A notable example is *CVE-2018-21269*, reported on January 24, 2018, in *issue#201*. Developers disagreed on whether certain usage patterns should be restricted. One contributor questioned the need for enforcement, saying, *"All we are doing is scanning the path and warning if there are any symbolic links"*. Another pushed for stricter handling and a deprecation period, suggesting, *"My vote is to ban them... and take that value from the upstream build system"*. Compatibility concerns also came up and a developer noted, *"What if, for example,* `/var` *is a symlink... Either* `/opt` *or* `/opt/package` *might legitimately be a symlink"*. The ongoing back-and-forth delayed progress, with public disclosure occurring on October 27, 2020, and a complete fix not applied until November 20, 2020, nearly three years after the initial report. We identified **8** cases where similar disagreements led to delays in fixing vulnerabilities.



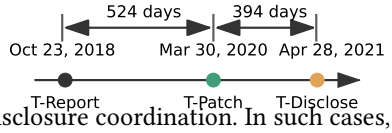
Category 6: Other or Unknown Cases. In certain cases, delays in vulnerability resolution cannot be attributed to any specific cause due to missing or unclear information. These situations arise when there are no recorded discussions, commits, or documentation explaining the time gap between the initial report (T_{Report}) and the patch (T_{Patch}), making it difficult to determine why the fix or disclosure was delayed. For instance, *CVE-2021-3639* was reported on July 9, 2021, and a fix was implemented on July 29, 2021. There were no recorded commits or discussions during this period to explain the delay. Public disclosure took place later, on August 22, 2022, more than a year after the patch was applied. Without clear evidence, it remains uncertain whether the delay was due to internal decisions, security concerns, or other unknown factors. Of the 94 PCVEs analyzed, **7** cases fall into this category.

Category 7: Resource Constraints. This category refers to delays in vulnerability resolution caused by limited development resources, such as time and contributor availability. Such constraints are common in open-source projects, which often depend on volunteer contributions and informal coordination. A clear example is *CVE-2020-15163* from *theupdateframework/python-tuf* , where progress was slowed due to a lack of available contributors. An initial patch was submitted on June 4, 2019, through *PR#885*, but the work remained incomplete for several months. On June 13, 2019, the project manager requested assistance with writing test cases, commenting, *"Maybe an intern"*. In a later comment on the same PR, they reiterated the request more explicitly: *"Do we have an intern who can take over, that is, fix the things I noted below and add some tests?"*. The tests were eventually added on October 3, 2019, and the final fix was merged on October 7, 2019, approximately four months after the initial call for help. Public disclosure occurred later, on September 9, 2020, nearly a year after the patch was completed. This example illustrates how even acknowledged vulnerabilities can face prolonged resolution times when contributors are unavailable. We identified **4** cases in which patch completion was delayed due to resource-related challenges, highlighting the need for sustained contributor engagement and structured collaboration in open-source security work.



Category 8: Lack of Expertise. In some cases, vulnerability resolution is delayed due to limited security expertise among project maintainers. This can result in uncertainty, extended discussions, and hesitation in applying a fix. Without a clear understanding of the issue or its implications, maintainers may postpone action, delaying both patching and disclosure. For example, *CVE-2020-22781* from *ether/etherpad-lite* was reported on October 23, 2018, but the patch was not implemented until March 30, 2020, and public disclosure followed over a year later on April 28, 2021. The delay was partly due to the absence of a formal reporting process and the maintainers' unfamiliarity with handling security issues. One maintainer openly stated,

Just Accepted



"I'm not clever enough to resolve this. I need someone with expertise to help!". This highlights a broader challenge in some open-source projects, where maintainers may lack experience in secure coding, vulnerability triage, or disclosure coordination. In such cases, progress often depends on support from external contributors or security researchers. **2** cases out of the 94 PCVEs involved delays linked to limited expertise, underscoring the importance of domain knowledge for timely vulnerability resolution.

Our analysis of PCVE resolution timelines shows that the longest delays often occur between implementing a fix and reporting it, with 65 cases involving delayed publication to the NVD. These gaps are difficult to explain due to the absence of documentation or recorded discussions between the fix (T_{Patch}) and the disclosure ($T_{Disclose}$) times. The second most common cause of delay is the misjudgment of a vulnerability's relevance or severity by maintainers, which can lead to deprioritization or extended debates over classification. Additional factors contributing to delays include organizational and technical challenges such as inactive maintenance, disagreements on appropriate fixes, and incomplete or ineffective patches. Limited resources and expertise also affect patch deployment, as many projects face contributor shortages, lack testing infrastructure, or require external support for security-related tasks. Collectively, these challenges point to broader issues in open-source vulnerability management, particularly inconsistent reporting practices and communication gaps that hinder timely and effective resolution.

Finding: We identify eight key factors that contribute to delays in PCVE resolution, with the most prevalent being (a) delayed reporting to the NVD, (b) misjudgment of vulnerability severity and relevance, and (c) lack of active project maintenance. To minimize these delays, developers should prioritize security vulnerabilities appropriately and ensure timely disclosure of fixes.

4 RQ2: To what extent do the SOTA vulnerability detection methods demonstrate efficacy in identifying PCVEs?

4.1 Data Preparation

As described in Sec. 2, we selected five SOTA methods as the baseline for evaluation, *LineVul* [31], *DeepDFA* [74], *VulCurator* [57], *MemVul* [62], and *PatchRNN* [77]. For RQ2, we selected a subset of 2,402 PCVEs released for 21 years (from 1999 to 2020).[1] To accommodate the diverse input data requirements of various SOTA methods (as summarized in Table 3, column 'Artifact type'), we organized the PCVEs accordingly to ensure a fair evaluation setup. For methods that require code-level inputs (e.g., diffs, functions, or code slices), we included only PCVEs whose patches or affected files are written in a supported programming language, specifically C, C++, or Java. Overall, we successfully collected 1,213 out of 2,402 PCVEs, each containing at least one of the necessary artifacts that can be used to evaluate one or more SOTA methods in a compatible programming language. In addition, we included non-vulnerability data points for each method to facilitate accurate comparisons. The detailed data collection procedure is described below.

4.1.1 ***MemVul.*** Utilizing a DL-based approach that incorporates language models and external vulnerability knowledge from the CWE, *MemVul* takes GitHub issues as input for vulnerability identification, focusing on the titles and bodies, as they were first created. Among the 2,402 PCVEs from our dataset, 1,059 PCVEs have at least one referenced issue, totalling 1,109 unique issues, all created before the NVD disclosure date of their corresponding CVEs. We removed the 399 issues that have been used for *MemVul*'s training purpose [62], resulting in a total of 683 PCVEs and corresponding 710 issues as summarized in Table 3, column '#Vuln'. Finally, we collected the titles and bodies of all the 710 issues at the time they were created from the GitHub Archive [4].

[1]PCVEs released from 2021 to 2024 are utilized for the evaluation of RQ3

Just Accepted

4.1.2 ***VulCurator****.* *VulCurator* leverages DL techniques to identify fixing commits for vulnerabilities through the analysis of issue–commit pairs. In particular, the issue title, body, and comments, along with the commit message and patch, are used. To extract issue–commit pairs in our dataset, we first extract issue IDs from commit messages using the same regular-expression matching method described in the *VulCurator* paper [58]. Further, to recover additional pairs, we leverage the issue timeline events to identify linked commits using the GitHub API [35].

Data filtering. Given our emphasis on gathering information prior to the NVD disclosure date to deduce the heuristics that might result in an earlier disclosure, we verify that both the issues and commits are created before the NVD disclosure date in all instances. To make a fair comparison with *MemVul*, which only accounts for information available at the time the issue was created, we also refine the issue–commit pairing by solely considering the information available when the issue–commit link was established. For example, if the issue is created before the commit, we extract its title, body, and comments as of the commit creation date. If the commit is created before the issue, we extract the issue title and body as of the issue creation date.

After applying these rules and restricting code-related artifacts to supported programming languages (C, C++, and Java), our dataset contains 586 commits, 198 issues, and 631 issue–commit pairs related to 210 PCVEs to evaluate *VulCurator*.

4.1.3 ***LineVul****.* The original *LineVul*, targeting C/C++, employs a transformer-based, fine-grained approach for predicting vulnerabilities and analyzing source code at the line level to identify potential security weaknesses [31]. Originally designed for C/C++, we extended *LineVul*'s functionality to support Java. The fine-tuning process involved constructing the training dataset and designing experiments, resulting in an F1 score of 0.65 across Java.

To obtain the modified C, C++, and Java source code files from our dataset, we initially pinpointed 486 PCVEs referenced with at least one commit. Subsequently, we extract a total of 534 commits, from which we gather 793 files, encompassing source files written in C, C++, and Java as detected using their file extensions (i.e., `.c`, `.cpp`, `.cxx`, `.java`). Further, as *LineVul* requires the source code files to be split into functions, we extract a total of 1,656 functions from the files by utilizing `SrcML` [19]. From these 1,656 functions, we identify the functions that are modified by the commits. We identified 1,656 modified functions linked to 486 unique PCVEs.

4.1.4 ***DeepDFA****.*. *DeepDFA* is a deep learning–based vulnerability detection model that enhances token-level representations such as CFGs and DFGs, enabling richer structural and semantic program analysis [74]. DeepDFA is designed for detecting vulnerabilities in C/C++ projects by integrating data-flow information directly into the learning process.

In our study, we reuse the dataset constructed for *LineVul* and restrict it to C/C++ source files to align with the original scope of *DeepDFA*. After filtering, the resulting dataset contains 407 unique PCVEs associated with 453 commits and 590 modified C/C++ files. Since *DeepDFA* operates at the function level, we further evaluate it on 1,212 modified C/C++ functions, for which we generate the required CFG and DFG representations.

4.1.5 ***PatchRNN****.*. *PatchRNN* is an RNN-based model designed to identify vulnerability-fixing commits by jointly encoding commit messages and patch diffs [77]. Unlike *LineVul*, which requires function-level representations, *PatchRNN* operates directly on commit-level artifacts, including the commit message and code.

During the data collection process, we retained only commits that (1) modify source files written in supported programming languages (C, C++, or Java), and (2) are created prior to the NVD disclosure date. Consequently, the *PatchRNN* dataset inherits these constraints and contains only commits for which valid diffs and corresponding file modifications were available. Following this filtering process, we obtained a total of 660 PCVEs associated with 1,199 commits. For each commit, we collect the commit message and the full patch diff, which constitute the required input to *PatchRNN*.

Just Accepted

Table 3. (RQ2) Data preparation results and the performance of SOTA methods on the PCVE dataset.

SOTA	Input Data			Evaluation Results							
	Artifact Type	#Vuln	#Non-Vul	TP	FP	FN	TN	Prec.	F1	Applicable _Recall	All _Recall
MemVul	Issue	683	549	**525**	273	276	**158**	0.66	0.71	0.77	**0.22**
VulCurator	Issue–Commit Pair	210	156	109	**84**	72	101	0.60	0.56	0.52	0.05
LineVul	Source Code Function	486	476	219	473	267	3	0.32	0.37	0.45	0.09
DeepDFA	Source Code Function	393	157	323	151	**70**	6	**0.68**	**0.75**	**0.82**	0.13
PatchRNN	Commit	660	616	444	496	216	120	0.47	0.56	0.67	0.18

4.1.6 ***Collection of Non-vulnerable artifacts.*** We constructed a dataset comprising non-vulnerable artifacts, maintaining a 1:1 ratio at the CVE level and a 1:5 ratio at the artifact level between vulnerable and non-vulnerable data, using the augmentation process described in previous studies [57, 58, 69]. For example, if a CVE is associated with one issue and two commits, we randomly sample five non-vulnerable issues and ten non-vulnerable commits to create a corresponding non-vulnerable data point. However, some repositories may not have sufficient artifacts due to limited activity, resulting in a discrepancy between the number of vulnerable and non-vulnerable data points. In particular, we employ a four-step method to gather non-vulnerable artifacts, including issues and commits. The following description uses commits as an example:

- *Step 1*: For each CVE referencing a GitHub commit, we collected all the commits from the corresponding repository.
- *Step 2*: We removed all the commits referenced by any CVE published between 1999 (the first vulnerability logged in the NVD) and 2020 by comparing them against the NVD database.
- *Step 3*: To ensure the non-vulnerable commit is created within the same time frame as the vulnerable artifact, maintaining relevance and timeliness, we only retained commits made within a six-month window surrounding the vulnerable artifact's creation date [69].
- *Step 4*: We randomly selected five commits from the retained list.

As such, for each SOTA method, we follow the same data preparation process to combine the collected non-vulnerable artifacts based on the corresponding CVEs, programming languages, and the specific model under evaluation. We obtained 2,109 non-vulnerable issues used for *MemVul*, producing 549 non-vulnerable data points. For *VulCurator*, we collected 429 commits and 803 issues, with 4,792 non-vulnerable issue–commit pairs, yielding 156 non-vulnerable data points. For *LineVul*, we collected 4,786 non-vulnerable source code functions and 383 commits, resulting in 476 non-vulnerable data points. For *DeepDFA*, after restricting to C/C++ projects, we collected 3,151 non-vulnerable C/C++ functions associated with 237 non-vulnerable commits, corresponding to 397 non-vulnerable data points. For *PatchRNN*, we collected 1,167 non-vulnerable commits, resulting in 616 non-vulnerable data points. The summary of the input data is presented in Table 3, column '#Non-Vuln'. The replication package includes our script for these steps [7].

4.2 Results

4.2.1 ***Overview of CVE Detection Rates.*** In Table 3, the column titled 'Evaluation Results' displays the outcomes of the evaluation. Given the diverse input data requirements of various SOTA methods, each applies to

Just Accepted

a subset of PCVEs for experimentation. As mentioned before, 1,213 out of 2,402 PCVEs are applicable to at least one of the SOTA methods. Therefore, we report the performance of the SOTA methods on our dataset using two recall metrics:

- Recall in **applicable** PCVEs per each model:

$$Applicable\_Recall = \frac{\#Detected\ PCVEs}{\#Applicable\ PCVEs}$$

- Recall in **all** PCVEs, where $\#All\ PCVEs = 2,204$:

$$All\_Recall = \frac{\#Detected\ PCVEs}{\#All\ PCVEs}$$

Our findings reveal that *DeepDFA* attained the highest applicable recall, achieving a recall value of 0.82 on the applicable dataset. In contrast, *MemVul* achieved the highest all recall, with a value of 0.22 across all PCVEs. Additionally, *DeepDFA* achieved the highest precision of 0.68 among the SOTA methods. *DeepDFA* also achieved the highest F1 score, indicating a strong balance between precision and recall within its supported language scope. Notably, compared to *LineVul*, which achieves an applicable recall of 0.45 and an F1 score of 0.37, *DeepDFA* demonstrates substantially stronger performance within the same source-code setting, suggesting that the integration of control-flow and data-flow representations contributes to improved detection effectiveness. These results highlight the distinction between applicable recall, which measures performance within a model's supported subset, and all recall, which measures coverage across the full PCVE set.

In total, 1,059 unique PCVEs are identified collectively by the SOTA methods after accounting for overlaps, leading to an overall all recall of approximately 0.44 over the 2,402 collected PCVEs. This collective recall exceeds the performance of any individual method, demonstrating that the models capture complementary subsets of vulnerabilities rather than detecting identical PCVEs.

Figure 4 illustrates the overlap in detection results among different SOTA methods. Specifically, *MemVul* detected 525 PCVEs out of a total of 2,402, including 450 unique PCVEs not identified by any other SOTA methods. *DeepDFA* uniquely detected 54 PCVEs, indicating additional but comparatively narrower unique coverage. *PatchRNN* identified 85 unique PCVEs, while *LineVul* detected 46 unique PCVEs. *VulCurator* identified 4 unique

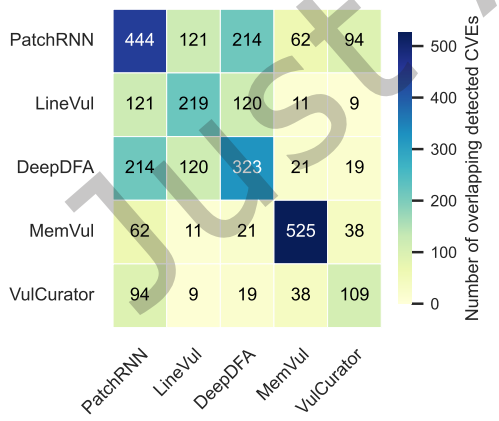


Fig. 4. Overlap in detection results among different SOTA methods.

Table 4. Additional iInformation sources for improving SOTA.

Source #	Information sources for vulnerability detection	#PCVEs
1	Keywords corresponding to CWE information	71 (63.96)
2	Manually linked artifacts offer additional information	44 (39.64)
3	Non-textual information offers additional information	36 (32.43)
4	GitHub PRs offer additional information	25 (22.52)
5	Manually assigned labels to issues or PRs provide crucial information	16 (14.41)

Just Accepted

PCVEs. These results further confirm that each approach specializes in identifying distinct vulnerability patterns and that measurable complementarity exists across methods.

Finding: SOTA methods exhibit different performance under two recall definitions across 1,213 applicable PCVEs. *DeepDFA* achieves the highest applicable recall and precision, while *MemVul* achieves the highest all recall across the full PCVE set. Collectively, the methods detect 1,059 unique PCVEs, achieving an overall all recall of approximately 0.44. The overlap analysis further indicates that each method specializes in identifying partially distinct subsets of PCVEs, suggesting that integrating these approaches could further improve overall PCVE detection rates.

4.2.2 ***Understanding the reason behind undetected PCVEs.*** Next, we qualitatively analyze the reasons behind the undetected PCVEs and derive insights for improving the performance of SOTA methods. Among the 1,213 applicable PCVEs, SOTA methods failed to detect 154. Similar to the qualitative analysis described in Sec. 3.3.1, for each PCVE, we analyze the lifecycle focusing on the information present in the GitHub artifacts. We performed stratified sampling by creating a bucket for each SOTA method using a confidence level of 95% and 5% margin of error, containing the set of PCVEs they failed to detect, resulting in 111 samples. We identified five types of information that could be useful for vulnerability detection, as summarized in Table 4.

Source 1: (63.96%) Keywords that correspond to CWE information. Our analysis revealed that out of 111 PCVEs, **71** PCVEs contain feature artifacts with keywords closely associated with CWE keywords. For example, when *CVE-2020-28471* from the *steveukx/properties* project was reported in *issue#40*, the description highlights a scenario involving "*prototype attributes being modified via a crafted properties file such that global object state becomes polluted*". In this case, the corresponding CWE-1321 (Improperly Controlled Modification of Object Prototype Attributes – Prototype Pollution) is semantically similar to the vulnerability report, which describes the attack mechanism. If utilized, the CVE could potentially be fixed and disclosed earlier. Similar instances can be found in *CVE-2020-22203*, where *issue#6*, titled "*phpcms2008 /yp/job.php genre parameter SQL inject*", is semantically similar to *CWE-89*, which describes improper neutralization of special elements in SQL commands leading to SQL injection vulnerabilities. **Insight:** One of the SOTA methods, *MemVul* has already leveraged the CWE information in conjunction with GitHub issues to detect vulnerabilities. Our results indicate that the incorporation of various other artifacts along with the CWE information can potentially improve the efficiency of vulnerability classification.

Source 2: (39.64%) Manually linked artifacts offer additional information. During the discussion of resolving vulnerabilities, OSS developers often manually link relevant GitHub artifacts for efficient communication. Such activity is supported by GitHub's cross-reference feature, including Autolinked references, URLs [2], and cross-referenced issue events [3]. Currently, amongst the SOTA only *VulCurator* utilizes linked artifacts by evaluating only the links between issues and commits. However, linkages between other types of artifacts not restricted to issue-commit links are useful for the identification of vulnerabilities. For example, we present the timeline of *CVE-2020-7693* in Figure 5. On February 11, 2019, it was reported in *issue#252* and later fixed on March 6, 2020 via *commit#c8c68e0*. On May 30, 2020, a developer created *issue#11076* to discuss the vulnerability's impact, as it causes a denial of service. This issue was cross-referenced with the original *issue#252* in the description,

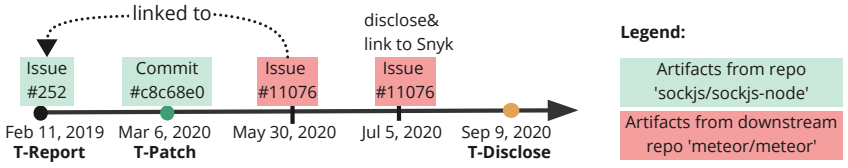


Fig. 5. Timeline of CVE-2020-7693.

Just Accepted

providing additional context about the vulnerability. On July 5, 2020, in the discussion thread, a developer added a link to Snyk, a security advisory, further indicating that the issue is a vulnerability. Similar example can be identified in *CVE-2016-9581*. In total, we identified **44** PCVEs in this category.

Insight: Prior studies have explored the cross-referenced links between code reviews and issues to interpret the intention behind changes [39, 48]. However, this data has not been employed for vulnerability detection. The linked artifacts provide a richer source of information for identifying vulnerabilities.

Source 3: (32.43%) Non-textual information. We discovered that developers use non-textual artifacts, such as images and proof-of-concept (POC) files, to communicate about vulnerabilities. Specifically, **36** PCVEs in this category included POC files or snapshots to replicate the vulnerability. These snapshots contained log traces and configuration details. For example, *CVE-2020-19720* from project *axiomatic-systems/Bento4* was reported in *issue#413*, in which the reporter provided a `poc_input4.zip` file and two annotated snapshots of the source code.

Insight: The investigation of identifying source code from non-textual sources, including images and videos, has been previously explored [60]. We posit that applying these techniques to enhance information for vulnerability detection holds significant promise.

Source 4: (22.52%) PRs offer additional information. The current SOTA methods do not include GitHub PRs for identifying vulnerabilities, even though PRs are similar to issues and contain discussions and commits, making their inclusion both intuitive and feasible. For example, *CVE-2017-17718* could be identified earlier if the corresponding *PR#259*, created for discussing and patching the vulnerability, were detected. We identified **25** out of 111 PCVEs that could have been detected earlier if their PR information had been included.

Source 5: (14.41%) Manually assigned labels to issues and PRs provide crucial information. GitHub supports the addition of labels to issues and PRs [43], which can indicate security-related information and facilitate early vulnerability detection. For example, *CVE-2019-16140* was reported on Jan 1, 2018 through *issue#2*. On Feb 2, 2018, the developer fixed the vulnerability in *commit#9e9f1fb* without any discussion. Later, a "bug" label was assigned on Jul 6, 2019. On Aug 31, 2019, a contributor pointed out that the issue appeared to be an exploitable use-after-free vulnerability and recommended filing a security advisory. Consequently, a "security" label was added on Sep 1, 2019. This sequence illustrates how manually applied labels later revealed the issue's security relevance. We observed similar patterns in our dataset. For example, security-related labels such as "cat:security" were added in *CVE-2018-25031* within *issue#4872* . In total, we identified **16** PCVEs exhibiting this label evolution pattern.

Insight: Prior work has shown that issue and PR labels are meaningful signals (Izquierdo et al. [43]; Kallis et al. [44]), including security-specific labels (Buhlmann et al. [13]; Palacio et al. [61]). These labels provide strong indicators of security relevance and are valuable input features, potentially with higher weighting.

Finding: Various types of resources can aid in detecting more vulnerabilities. These include the similarity between issue discussions and CWE keywords, cross-referenced artifacts slightly removed from directly related information, PR-related information, developer-assigned labels, and non-textual information. Future research could enhance vulnerability detection performance by integrating these diverse artifacts.

Just Accepted

5 RQ3: How can the SOTA be improved to identify PCVEs that currently remain undetected?

5.1 Research Method

This section utilizes the findings from RQ2 to create an improved method for detecting a greater number of PCVEs. We restrict our analysis to text-based artifacts, intentionally excluding non-textual elements like screenshots and attached files, considering their complexity and encouraging future exploration. We define our method as

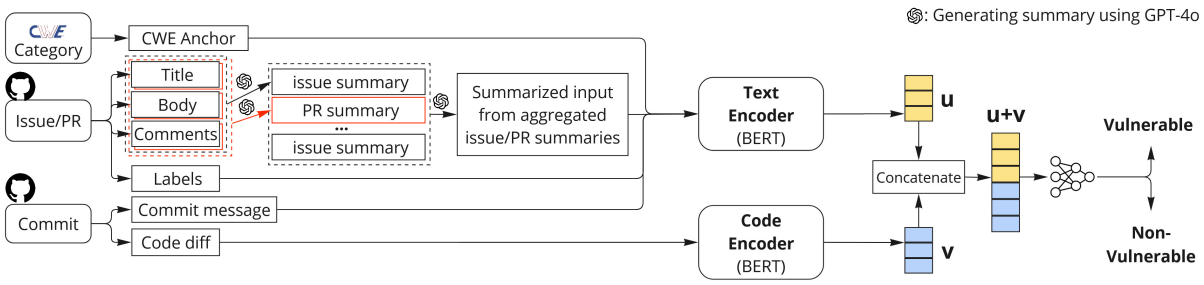


Fig. 6. Architecture of *DeeptraVul*.

DeeptraVul, which is short for **De**t**e**cting **P**ro**tra**cted **Vul**nerabilities. We first introduce the overall architecture of *DeeptraVul*, followed by the model training and inference details. We present the architecture in Figure 6.

5.1.1 ***Input Representation.*** There are two types of artifacts used in *DeeptraVul*, including *text-based* input and *code-based* input.

Text-based input. In addition to the artifacts that are covered by the SOTA methods, as presented in Table 3 column 'Artifact type', we added four new types of artifacts as input, including (1) PRs, (2) labels of both issues and PRs, (3) linked artifacts in the issue/PR timeline events [35], and (4) CWE Anchor, a representation of the CWE category-related information reused from prior work [62]. A PCVE is included in our dataset when it includes at least one commit and one issue or PR, ensuring that every case contains both code evidence and its accompanying development context.

We used the Bidirectional Encoder Representations from Transformers (BERT) architecture to convert text-based input, however, it only supports encoding up to 512 tokens. To meet the token constraint and keep as much information as possible, we designed a two-step summarization using *GPT-4o* [59]. Earlier research has shown that GPT models perform well in summarization tasks, including the summarization of both generic and vulnerability focused reports [10, 17, 54, 65, 67, 79]. More recently, models in the GPT-4 class have been adopted in empirical studies because they offer stronger performance than GPT-3 models and have been evaluated for a variety of security analysis tasks such as vulnerability detection, classification, repair, and broader security assessment [14, 32]. They have also been used to summarize vulnerability reports and other security relevant technical content [18]. These developments support our choice of *GPT-4o* for preprocessing long textual artifacts.

In our pipeline, we (1) summarize each GitHub issue and PR individually, including their titles, bodies, and comments, then (2) aggregate these summaries to produce a consolidated overall summary. The prompts used for both steps are presented in Figure 7.

Finally, we leveraged BERT, trained with the Robustly Optimized BERT Pretraining Approach (ROBERTA) [24, 53], incorporating training weights from *MemVul*. In this context, the ROBERTA encoder was fine-tuned for vulnerability prediction using textual data sourced from GitHub issues [62].

Code-based input is the code `diff` of each commit, including both the referenced commit in NVD, commits included in PRs, and the linked commits in issue/PR discussion threads. We leveraged CodeBERT (a variant of BERT) to create the feature vectors [28]. Regarding the neural network classifier, we use a one-layer neural network classifier for the classification task.

5.1.2 ***Data collection & preprocessing.*** We gathered relevant artifacts from each CVE from three sources: (a) GitHub artifacts listed in the NVD References section, such as issues, PRs, and commits created prior to disclosure; (b) cross-referenced artifacts derived from timeline events of issues and PRs; and (c) artifacts connected through hyperlinks found in the text of (a) using the same pattern matching as prior work [58]. Similarly, we construct a

Just Accepted

```
Objective (O): We are building a deep learning system for vulnerability
classification. You are tasked with summarizing GitHub Issues and Pull
Requests. The summary should be informative and technical.

Style (S): Capture key discussion points.
Tone (T): Reflect sentiment present in the discussion.
Audience (A): A classification model.
Constraint: Ensure no source code appears in the summary.

Input:
<TITLE>
<BODY>
<COMMENTS>
```

(a) Step 1 prompt used to summarize each Issue or PR, including title, body, and comments.

```
Summarize the text concisely and ensure the summary is brief and strictly
to the point, using as few characters as possible.
Ensure there are no source code components in the summary.

Input:
<STEP_1_SUMMARY_COLLECTION>
```

(b) Step 2 prompt used to generate the aggregated summary.

Fig. 7. Two-step summarization prompts used prior to *DeeptraVul* encoding.

Table 5. Summary of Training and Evaluation Data used for *DeeptraVul* by Programming Language.

Artifact	Vulnerability Status	Programming Language						Total Train	Total Eval
		C		C++		Java			
		Train	Eval	Train	Eval	Train	Eval		
Issue	Vuln	194	64	47	30	33	6	274	100
	Non-Vuln	611	249	148	88	104	17	863	354
Commit	Vuln	605	250	175	93	187	57	967	400
	Non-Vuln	1847	747	410	284	558	143	2815	1174
PR	Vuln	98	48	39	17	55	23	192	88
	Non-Vuln	325	178	95	56	173	66	593	300

collection of non-vulnerability datasets for training and evaluation following the procedure in Sec. 4.1.6. In Table 5, we present the dataset statistics per programming language. We further split the full dataset into 80% training, 10% validation (covering years 1999–2020), and 10% testing (covering years 2020–2024). *DeeptraVul* achieved an average F1 score of 0.84, precision of 0.84, and an applicable recall of 0.85 across the training and validation sets.

5.1.3 ***LLM-Based Detector (Chat*GPT-4o*).*** To complement neural encoders, we investigate whether modern reasoning-capable LLMs can detect vulnerability signals directly from heterogeneous development artifacts rather than relying on engineered representations. Prior studies report that LLMs can infer latent security patterns, reason over causal relationships in developer discussions, and surface vulnerability indicators that traditional

Just Accepted

```
You are a security analysis assistant.
Identify whether the provided artifacts indicate a security vulnerability.

Respond strictly in this format:
Answer: Yes / No
Justification: (one concise sentence explaining why)

=== PULL REQUEST DESCRIPTION ===
<PR text, if available>

=== ISSUE DESCRIPTION ===
<Issue text, if available>

=== COMMIT MESSAGE ===
<Commit message, if available>

=== COMMIT DIFF ===
<Commit diff, if available>
```

Fig. 8. Prompt used to evaluate *GPT-4o* on consolidated PCVE artifacts.

token-based models may fail to capture [14, 16, 80]. Thus, evaluating an LLM alongside conventional baselines enables us to assess whether such reasoning benefits extend to PCVE detection.

To ensure comparability with *DeeptraVul*, we adopt an artifact preparation process consistent with our pipeline, with the exception that CWE anchor information is omitted because the LLM processes raw linguistic cues directly. All associated evidence, including issue text, PR descriptions, commit messages, and pre-disclosure patch diffs, are combined into a single consolidated instance per CVE. As in *DeeptraVul*, artifacts are limited to supported source languages (C, C++, and Java) when code content is present.

Chat*GPT-4o* is used in a zero-shot classification setting [59]: for each consolidated artifact bundle, the model is asked to determine whether it indicates a security vulnerability. A fixed prompt, as shown in Figure 8, produces a binary decision (“Yes’’ / “No’’) similar to other baselines, enabling direct comparison.

5.1.4 ***Evaluation Setup.*** The evaluation dataset for *DeeptraVul* comprised 235 PCVEs from 2021 until 2024. For comparative analysis with the SOTA methods, we included additional relevant artifacts as input for other models. In particular, *MemVul* was evaluated using 3,672 issue/PR records yielding 524 vulnerable and 449 non-vulnerable data points. Similarly, *VulCurator* was evaluated using 7,325 issue-commit and PR-commit pairs, *LineVul* processed 430 commits containing valid function data, *DeepDFA* processed 305 commits containing valid function data, and *PatchRNN* processed 1,340 commits with required filtering and language constraints. Table 6 provides the vulnerable and non-vulnerable counts for each baseline. Our experiments were executed on a CentOS 7 environment using an NVIDIA A100-40GB GPU.

5.2 Results

In Table 6, we summarize the performance of the SOTA methods, *DeeptraVul*, and the GPT model. As in RQ2, the baselines remain constrained to the artifacts they support.

Across 826 PCVEs, the baseline detectors are applicable on different PCVE subsets, as vulnerability evidence appears unevenly across commits, PRs, issues, and related artifacts. After applying artifact and language constraints, *PatchRNN*, *LineVul*, *DeepDFA*, *MemVul*, and *VulCurator* are applicable to 253, 157, 125, 524, and 150 CVEs respectively. *DeeptraVul* and *GPT-4o* apply to 134 CVEs and contribute complementary coverage. Collectively, the models detect 486 of the 826 PCVEs, corresponding to an overall coverage of 0.58 within the constrained RQ3 setting.

Just Accepted

Table 6. (RQ3) Data preparation results and the performance of SOTA methods, *GPT-4o* and *DeeptraVul* on the PCVE dataset.

SOTA	Input Data			Evaluation Results							
	Artifact Type	#Vuln	#Non-Vul	TP	FP	FN	TN	Prec.	F1	Applicable _Recall	All _Recall
MemVul	Issue + PR	524	449	**285**	138	239	**311**	0.67	0.60	0.54	**0.35**
VulCurator	Issue-Commit + PR-Commit	150	121	75	70	75	51	0.52	0.51	0.50	0.09
LineVul	Source Code Function	157	154	75	148	82	6	0.34	0.40	0.48	0.09
DeepDFA	Source Code Function	125	71	109	68	16	3	0.62	0.72	0.87	0.13
PatchRNN	Commit	253	277	181	191	72	36	0.49	0.58	0.72	0.22
DeeptraVul	Issue + PR + Commit	134	101	121	**28**	**13**	73	**0.81**	**0.86**	**0.90**	0.15
LLM (GPT-4o)	Issue + PR + Commit	134	101	112	63	22	38	0.64	0.72	0.84	0.14

Within this applicable subset, detection outcomes vary substantially across models. *MemVul* detects 285 PCVEs, *PatchRNN* detects 181, and both *LineVul* and *VulCurator* detect 75 PCVEs. *DeepDFA* detects 109 PCVEs with an applicable recall of 0.87, demonstrating strong performance within its constrained source-code setting. Compared to *LineVul*, *DeepDFA* identifies a larger portion of vulnerabilities under the same language restrictions, reflecting the benefit of incorporating structural program information when analysis is limited to source code artifacts. *DeeptraVul* detects 121 cases, while *GPT-4o* detects 112. These differences in detection volume are also reflected in precision and F1 performance. *DeeptraVul* achieves a precision of 0.81 and an F1 score of 0.86, followed by *GPT-4o* at 0.64 precision and 0.72 F1, while *DeepDFA* attains 0.62 precision and 0.72 F1. Overall, the results underscore meaningful differences in selectivity and coverage across detectors, with *DeeptraVul* providing the most balanced trade-off between detection breadth and accuracy within its applicable scope.

5.2.1 ***Generalization Across Programming Languages.*** Table 7 reports language-level detection effectiveness for the five evaluated models that expose language metadata. Overall, the results show substantial variation in performance across programming languages.

DeeptraVul achieves the strongest detection capability for C with a recall detection score of 0.91 and for C++ with a score of 0.95, outperforming all other models on native-code languages, while maintaining moderate effectiveness for Java at 0.54. *GPT-4o* ranks second in terms of balanced performance, with high detection effectiveness for C at 0.89 and C++ at 0.82, and comparatively stronger performance on Java at 0.63. *PatchRNN* exhibits a similar trend, achieving strong effectiveness for C at 0.77 and C++ at 0.80, but showing a pronounced decline for Java at 0.33, indicating that its learned representations generalize more effectively to vulnerability patterns common in low-level languages, consistent with its predominantly C and C++ training data. *DeepDFA* achieves strong effectiveness on C++ at 0.92 and lower performance on C at 0.31. Since *DeepDFA* was originally designed and evaluated for C and C++ code, it is applied here within its supported language scope and therefore does not report results for Java.

In contrast, *LineVul* and *VulCurator* demonstrate a different pattern. *LineVul* achieves its strongest effectiveness on Java at 0.82, followed by C++ at 0.78, while its performance on C at 0.33 is substantially lower. Similarly, *VulCurator* attains its highest effectiveness on Java at 0.72, followed by C++ at 0.49 and C at 0.42.

Just Accepted

These findings highlight systematic differences in cross-language generalization, with *LineVul* and *VulCurator* performing better on Java, and *GPT-4o*, *PatchRNN*, *DeepDFA*, and *DeeptraVul* exhibiting stronger performance on C and C++. **Language sensitivity emerges as a consistent factor in model performance, shaped by architectural decisions, training composition, and language-specific vulnerability patterns. No detector exhibits universally strong behavior across languages.**

5.2.2 ***Review of Misclassification Cases.*** The observed variability in detector performance across languages extends beyond aggregate language-level metrics and is reflected in the models' decision behavior. Specifically, the same architectural and data-driven factors that shape language sensitivity also influence how models balance coverage and reliability. Although the baseline detectors, such as *MemVul* and *PatchRNN*, identify more CVEs in absolute terms, this is largely because they rely on a single artifact type. *PatchRNN* uses commits and *MemVul* uses issues, whereas *DeeptraVul* requires the presence of at least one commit linked to an issue or a PR, which naturally reduces the number of detectable CVEs. Consequently, approaches that rely on a single artifact type exhibit higher false positive and false negative rates. By comparison, both *DeeptraVul* and *GPT-4o* produce markedly fewer false positives, indicating that they learn more selective and accurate decision boundaries. This behavior suggests that the reasoning processes of *DeeptraVul* and *GPT-4o* are more effective at distinguishing true vulnerability evidence from non-security signals, even when their overall detection volumes differ.

Therefore, to enable a fair comparison, we restrict the evaluation to the subset of 134 PCVEs for which *DeeptraVul* was applicable and for which all models had access to the necessary artifacts. Within this controlled setting, the performance distinction becomes clearer: *DeeptraVul* detects 121 PCVEs and *GPT-4o* detects 112 PCVEs, while the baseline methods detect fewer cases within this PCVE subset.

Figure 9 visualizes these overlaps, illustrating both shared coverage and remaining gaps across detectors. This contrast shows that *DeeptraVul*'s advantage is most apparent when all models are required to reason over the same evidence rather than over their preferred artifact types. To understand the remaining gaps, we analyse

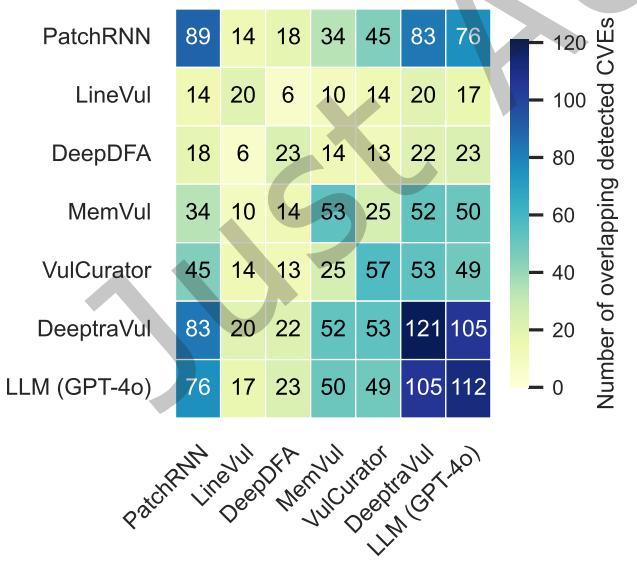


Fig. 9. Overlap in detection results on *DeeptraVul* dataset.

Table 7. Language-level PCVE detection effectiveness across vulnerability detectors.

Model	C	C++	Java	Overall
VulCurator	0.42	0.49	0.72	0.50
LineVul	0.33	0.78	0.82	0.48
DeepDFA	0.38	0.92	-	0.87
PatchRNN	0.77	0.80	0.33	0.72
DeeptraVul	0.91	0.95	0.54	0.90
GPT-4o	0.89	0.82	0.63	0.84

Table 8. *DeeptraVul* AUC performance under different artifact configurations.

Features	AUC	ΔAUC
Code	0.49	+0.32
Issue + PR	0.64	+0.17
Commit (Msg + Diff)	0.66	+0.15
Issue + PR + Commit Msg	0.78	+0.04
All Features	0.81	—

Just Accepted

the cases *DeeptraVul* failed to detect. Eight of these involve non-textual artifacts, such as compressed test suites, proof of concept exploit files, or screenshots highlighting affected code regions, that are not currently supported by the pipeline. **This observation suggests that extending the model to handle multi-modal inputs could recover a notable portion of these missed detections.**

Four of the remaining cases point to limitations in evidence availability rather than model capacity alone. Two of these were also missed by *GPT-4o*, indicating that the underlying artifacts provide limited or ambiguous security cues. To probe this ambiguity, we previously asked *GPT-4o* to justify its predictions, as illustrated in Figure 8. The LLM response shows a recurring pattern in which corrective changes are presented as benign stability or feature improvements, masking the fact that their existence reflects a prior vulnerability. For example, *CVE-2021-4289* in *PR#89* introduces encoding of the `AppId` field to prevent XSS, which the model summarized as *"addressing an XSS vulnerability"*. The change was interpreted as a positive remediation action rather than as vulnerability-related evidence. Likewise, *CVE-2022-4963* in *PR#39* replaces unsafe SQL construction with prepared statements and was summarized as a maintenance improvement, and treated as not vulnerability-related. In both cases, positive framing in the model summaries led *DeeptraVul* and *GPT-4o* to infer quality improvement rather than security relevance. **Security-motivated code changes may not identified as vulnerability-related when described using positive, maintenance-oriented language.**

The final *DeeptraVul* miss illustrates a different issue. A representative example is *CVE-2022-21657*, in *commit#630*, where routing and configuration logic is restructured across multiple files with little explanatory text, as in commit messages. Although the vulnerability corresponds to CWE 295 (Improper Certificate Validation), vulnerability evidence exists but is embedded in many code edits, demonstrating that the presence of changed code does not necessarily translate into accessible narrative signals for automated reasoning. This motivates a need to investigate which artifact types and representations most influence *DeeptraVul*'s behaviour. Accordingly, we analyze how different artifact types and representation choices affect *DeeptraVul*'s ability to surface actionable vulnerability evidence.

5.2.3 ***Evaluation of* DeeptraVul *Under Different Settings.*** To examine how artifact availability influences detection performance, we evaluated *DeeptraVul* under several feature configurations. These included: (1) code-only input, following the setting of *LineVul* and *DeepDFA*; (2) issue and PR descriptions, as used by *MemVul*; (3) commit messages paired with code diffs, consistent with *PatchRNN*; (4) issue, PR, and commit message texts combined; and (5) a full configuration integrating issue descriptions, PR descriptions, commit messages, and code changes.

Insight 1: Multi-artifact evidence is critical for performance. On average, removing artifact types reduces AUC by approximately 27% relative to the full-evidence configuration, indicating the importance of integrating information from multiple sources. As shown in Table 8, the full configuration achieves the highest performance, with an AUC of 0.81.

Insight 2: Code-only evidence is insufficient. The code-only configuration yields an AUC of 0.49, representing a decline of roughly 40%, indicating that diffs alone provide insufficient context for distinguishing vulnerable from benign updates.

Insight 3: Textual context substantially improves detection. Issue and PR descriptions yield an AUC of 0.64, a reduction of 21% relative to full evidence, while commit messages paired with diffs reach 0.66, a 19% reduction. Combining issue, PR, and commit text achieves an AUC of 0.78, only 4% below the full configuration, capturing most contextual signal even without code changes.

Overall, these results show that *DeeptraVul* achieves its strongest performance when multiple artifact types are available, and that removing any single artifact degrades detection performance in predictable ways.

Just Accepted

Finding: *DeeptraVul* achieves the strongest overall detection performance among evaluated models, with 0.81 precision, 0.90 recall, and an F1 score of 0.86. Its effectiveness remains stable across C and C++, and its feature evaluation indicates that performance improves when multiple artifact sources are provided. Our qualitative analysis also suggests that supporting non-textual artifacts could further reduce the remaining missed PCVEs.

6 Discussion

6.1 For SE Researchers

This section highlights key areas where software engineering researchers can investigate and mitigate delays in the vulnerability lifecycle. The findings are also relevant to SE practitioners aiming to improve practices related to vulnerability reporting, detection, prioritization, and resolution.

(1) **Delays in CVE Reporting.** Timely disclosure of vulnerabilities is essential to minimize exposure and allow users to apply patches immediately. Our analysis shows that **46.27%** of PCVEs experienced delays during the NVD disclosure process, leaving systems vulnerable even after the fixes were implemented.
Many relevant artifacts, such as reasons for delays or related discussions, were missing from the NVD references. As a result, it becomes difficult to trace the full vulnerability timeline and understand the decisions behind these delays. For example, *CVE-2018-10757* was patched in *commit#c89158e* on April 26, 2015, but the CVE was not published until May 5, 2018, over three years later. The NVD lists the patch reference, but provides no explanation or metadata to clarify the delay.
In other cases, the delays were more transparent. *CVE-2020-35132* was delayed due to confusion around the reporting process. In the related *issue#130*, a contributor asked if a CVE had been assigned. The reporter replied: *"I don't think so — do you know how I can go about this? I apologize for the lack of knowledge / context here"*. The reporter also mentioned submitting a private ticket on Launchpad but was unsure how to request a CVE. Another contributor noted the time it takes to gather the necessary information and the risk of duplicate assignments. This shows how limited familiarity with the CVE process can unintentionally delay disclosure, even when issues are reported responsibly. This highlights the need for automated tools that monitor repository activity, reconstruct vulnerability histories, and flag disclosure delays.
To better understand these delays, it is useful to identify the common characteristics of CVE (i.e., patterns of delayed CVEs) that are associated with delays. For example, certain categories of CVEs may be more prone to delays due to their complexity, reporting practices, or lack of automation support. Understanding these patterns can inform the design of tools for earlier detection and intervention. Below are examples of delayed CVEs that indicate common patterns of delays.
    (a) **Memory corruption issues**, such as *CVE-2016-9581*, may involve complex debugging and verification, which can slow down disclosure.
    (b) **XSS vulnerabilities**, like *CVE-2020-18475*, may be underreported due to lack of proper documentation or awareness.
    (c) **Authentication flaws**, such as *CVE-2020-19268*, often suffer from insufficient automation and visibility, contributing to delays.
Beyond technical classifications, the reporting ecosystem itself plays a critical role in shaping disclosure timelines. Informal sources such as mailing lists and issue trackers often provide essential context missing from official databases. For instance, *CVE-2017-20005* was discussed in *Nginx issue #1368* and later referenced in the *Debian LTS mailing list*. Tools such as MITRE's CVE Services API [66] and GitHub Security Advisories [33] support structured submission workflows, but broader adoption and better integration are needed to improve the timeliness and completeness of disclosures.

(2) **Finer-Grained Analysis of Early Vulnerability Detection.** In line with the need for transparency in disclosure timelines, our study did not investigate the exact timing of artifact creation relative to vulnerability

Just Accepted

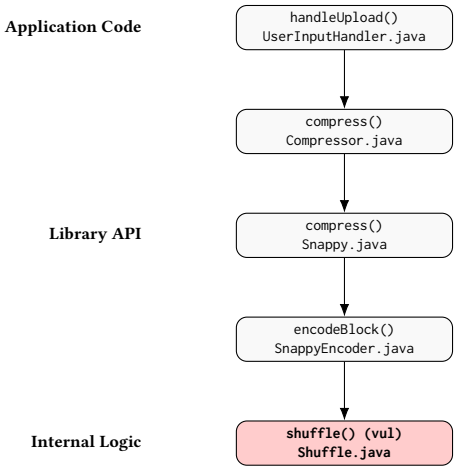


Fig. 10. Function-level dependency chain showing indirect reachability to the vulnerable `shuffle()` method (CVE-2023-34453). Layer labels on the left indicate architectural separation, while boxes show the function path.

resolution. This limits our understanding of when vulnerabilities are first detected. Future work should focus on reconstructing CVE timelines at a finer granularity to assess how early existing tools and techniques can detect vulnerabilities. Some vulnerabilities in our dataset remained unresolved across multiple release cycles, suggesting earlier detection was possible but missed. Leveraging historical repository data with predictive modeling may help identify vulnerabilities earlier. Time-series analyses across ecosystems could uncover patterns in how vulnerabilities emerge, enabling the development of early-warning systems that support more proactive detection and response strategies.

(3) **Vulnerability Prioritization.** Delays are not solely the result of external reporting gaps. Internal project decisions also play a role. Our data shows that **19.4%** of delayed PCVEs were due to *misjudgment of severity*, where vulnerabilities were deprioritized or overlooked. For example, *CVE-2018-10727* remained unresolved despite active development activity, while *CVE-2020-36420* involved prolonged use of a known vulnerable package. A developer's comment in a discussion *thread*, *"It would've been appreciated if you could've given us a heads up / let us in security@ and treecleaner@ know about your intentions"*, underscores the lack of internal visibility. To address this issue, development workflows should incorporate automated risk scoring systems based on objective security metrics [63]. Such tools can help teams identify and prioritize critical vulnerabilities more effectively, reducing delays caused by oversight or misclassification.

(4) **Investing in Tools for Patch Quality.** Even when vulnerabilities are detected and prioritized, the patching process itself can contribute to delays. In **7.64%** of PCVEs, initial fixes were incomplete and required multiple revisions. For example, CVE-2020-18705 required follow-up fixes less than two months after the initial patch in *issue#676*. These cases often result from time pressure, limited testing, or insufficient peer review. To mitigate such delays, future research should explore automated patch validation tools that help identify incomplete fixes before deployment. Techniques that improve first-attempt patch success rates can streamline the remediation process and reduce the window of exposure caused by ineffective patches.

(5) **Leveraging Image-Based Code Analysis for Vulnerability Detection.** Most vulnerability detection techniques rely on static, text-based code analysis. However, these approaches can miss important structural patterns that are easier to detect visually. Image-based code analysis provides a promising alternative by converting source code into visual formats, such as control-flow graphs or architectural diagrams, and

Just Accepted

applying computer vision techniques like convolutional neural networks (CNNs). When combined with traditional static analysis, these methods can improve detection accuracy and reduce false positives. For example, visualizing data flow or function-level dependencies may reveal vulnerabilities spread across multiple files that text-based tools might miss. Consider CVE-2023-34453, which involved an integer overflow in the `shuffle(int[] input)` method within the snappy-java library. Although this method might not be directly invoked in application code, its reachability can still pose a risk. For example, function-level dependency analysis could reveal that this method is indirectly reachable via internal library calls triggered by standard compression routines. While text-based scans may overlook such indirect paths, a visual dependency graph can help uncover how user-controlled input could propagate through these internal layers and eventually reach the vulnerable method. As illustrated in Figure 10, such a graph can make hidden call relationships explicit, highlighting both direct and indirect paths to vulnerable functions.

Our review of CVE-related GitHub repositories shows that visual artifacts are already used in practice. Many vulnerability reports include screenshots to demonstrate how the issue occurs or to point out the exact vulnerable functions. For instance, *CVE-2017-17480*, discussed in *issue#1044* of the *uclouvain/openjpeg* repository, contains images that help clarify the vulnerability. These findings highlight the value of incorporating visual information into automated vulnerability analysis. Using such visual context can make vulnerability reports more understandable and support the development of more accurate and practical detection tools.

(6) **Process Standardization and Ecosystem Comparisons.** Finally, delays often stem from inconsistencies in reporting workflows across software ecosystems. Manual triage by CVE authorities contributes to reporting bottlenecks, particularly in resource-constrained projects. Future research should investigate semi-automated classification methods to support faster CVE processing. A comparative study of proprietary and open-source ecosystems could help identify process models that promote timely reporting. Practices from structured proprietary pipelines may be adapted to open-source environments. Additionally, standardizing reporting frameworks to account for team structure, project scale, and communication patterns can improve consistency in disclosure practices. Improving coordination between development and security teams and integrating timelines into issue tracking can further reduce delays and enhance vulnerability response efforts.

6.2 For OSS Practitioners And Developers

This section provides actionable strategies for open-source practitioners and maintainers to improve vulnerability management and reduce delays in patching and disclosure. The recommendations are informed by recurring patterns observed in delayed CVEs.

(1) **Improving Vulnerability Identification Practices.** Prior studies label components without CVE assignments as non-vulnerable, although this does not confirm that the code is safe. Croft et al. show that real-world datasets lack ground-truth labels for non-vulnerable code and therefore cannot validate their correctness [20]. Similarly, *CVEfixes* collects CVE-linked vulnerable code but does not verify non-vulnerable samples [11]. This limitation also appears in VCMATCH, where negative samples are randomly selected commits treated as non-vulnerable [76]. Likewise, *DiverseVul* labels unchanged and post-patch versions of modified functions as non-vulnerable based on fixing commits, without validating these functions [15]. These practices show that non-vulnerable code identification relies on heuristics that may not hold.

For OSS practitioners and developers, these findings highlight the importance of avoiding implicit assumptions that unreported or unpatched code is non-vulnerable. Practitioners are encouraged to treat vulnerability status as evolving and incomplete, particularly for widely reused components and dependencies. Maintaining clear documentation of security-relevant changes, adopting structured vulnerability reporting practices, and proactively auditing critical or high-impact code paths can help reduce reliance on informal heuristics. Furthermore, OSS projects may benefit from explicitly distinguishing between code that has been reviewed

Just Accepted

for security and code whose vulnerability status remains unknown, thereby improving transparency for downstream users and integrators.

(2) **Enhancing Security Expertise Through Training and Reviews.** Delays in addressing vulnerabilities often stem from misjudgments in severity. Developers may fail to prioritize or properly address security issues due to limited expertise. For example, *CVE-2021-23418* remained unresolved in *Glances* despite active development, and *CVE-2020-27794* experienced delays due to prolonged discussion without resolution. To mitigate such cases, developers should participate in regular security training, use AI-assisted auditing tools, and engage in periodic project-wide security reviews. Peer-reviewed security assessments can also help improve classification accuracy and reduce delays caused by misjudgment.

(3) **Incentivizing Security Contributions to Address Resource Constraints.** A lack of dedicated contributors is another common barrier to timely patching. For instance, *CVE-2020-15163* in *python-tuf* faced delays due to contributor shortages. Approximately **3%** of PCVEs were affected by such workforce limitations. Open-source communities should introduce incentives such as security-focused bug bounties, mentorship programs, and recognition systems to attract contributors. Automated security checks in PRs can also streamline the review process.

(4) **Strengthening Security Policy Enforcement and Public Disclosure.** Some vulnerabilities are fixed silently without proper public disclosure, as in the case of *CVE-2017-7495*. Others, like *CVE-2021-32265*, were disclosed long after being patched. Projects should enforce structured security policies that mandate public disclosure of all security-related fixes. This includes integrating compliance checks into CI/CD pipelines and defining clear timelines and communication procedures for disclosure.

(5) **Ensuring Timely Patch Deployment Through Rigorous Validation.** Patches that undergo multiple revisions delay vulnerability resolution. An example is *CVE-2020-26299*, which took over 500 days to fully resolve. Projects should implement rigorous validation practices, including mandatory peer reviews, automated test coverage for patches, and post-deployment monitoring of patch effectiveness.

(6) **Mitigating Risks from Under-Maintained and Abandoned Projects.** Inactive or abandoned projects often leave vulnerabilities unpatched, as seen in *CVE-2016-11014* in *Disclosed*. Projects should designate backup maintainers and integrate tooling that flags unmaintained repositories with known vulnerabilities. Package managers and dependency platforms can assist by warning users when using outdated or insecure libraries.

6.3 For CVE Authorities (e.g., NVD)

(1) **Streamlining Vulnerability Reporting.** Delays in CVE reporting increase exposure risk. Although GitHub recommends security policy setup, adoption remains inconsistent. *CVE-2020-22781* offers a positive example, where *ether/etherpad-lite* encouraged private reporting through a dedicated channel [27]. CVE authorities should enforce standardized reporting workflows, clearly define submission responsibilities, and provide tools to help projects report vulnerabilities efficiently.

(2) **Enhancing Tooling Support.** High-recall detection techniques often result in developer fatigue due to false positives. Detection algorithms should be refined to balance precision and recall. Integrating triage automation into CVE pipelines would expedite classification and reduce manual workload.

By improving workflows, promoting transparency, and expanding tooling support, CVE authorities can significantly enhance the overall responsiveness of the vulnerability disclosure ecosystem.

Just Accepted

Finding: Delays in vulnerability disclosure often arise due to unclear communication, missing context, and limited automation. To address this:

(1) Researchers should investigate these causes and develop improved detection techniques, automated severity scoring, and early patch validation tools.

(2) OSS developers require stronger security training, AI-assisted reviews, clearer disclosure practices, and better dependency and maintenance management.

(3) CVE authorities should standardize reporting workflows, provide secure submission channels, and automate triage to improve consistency and response times.

7 Conclusion

In this study, we investigated the issue of PCVEs and analyzed the reasons behind delayed patching and disclosure. Through a qualitative analysis, we identified **eight** key causes contributing to these delays: lack of active maintenance, misjudgment of relevance and severity, disagreement on resolution, incomplete or insufficient fixes, resource constraints, lack of expertise, delayed NVD disclosure, and unknown causes.

To assess the effectiveness of existing vulnerability detection techniques, we evaluated SOTA methods on a curated PCVE dataset. Our results indicate that these methods successfully detect only **44%** of the instances, demonstrating their limitations in identifying protracted vulnerabilities. This suggests that conventional models, which typically rely on a single artifact, fail to capture the complexities of PCVEs. The results emphasize the need for more comprehensive approaches that incorporate multiple artifacts to improve detection accuracy.

To address these limitations, we introduce *DeeptraVul*, a model designed to enhance vulnerability detection. Unlike traditional approaches, *DeeptraVul* integrates multiple software artifacts, to provide a richer context for analysis. Our experimental results show that *DeeptraVul* significantly outperforms existing SOTA methods, including recent LLM-based approaches, achieving a higher detection rate on a subset of PCVEs. These findings highlight the importance of leveraging multiple artifacts for better vulnerability detection, ultimately aiding in the timely identification and mitigation of security risks.

Just Accepted

Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC): [RGPIN-2021-03538] and [RGPIN-2021-03969].

References

[1] 2018. ISO/IEC 29147:2018: Security techniques - Vulnerability disclosure. https://www.iso.org/standard/72311.html

[2] 2024. Autolinked references and URLs. https://docs.github.com/en/rest/using-the-rest-api/issue-event-types?apiVersion=2022-11-28#cross-referenced. Accessed: 2024-06-03.

[3] 2024. cross-referenced issue event type. https://docs.github.com/en/get-started/writing-on-github/working-with-advanced-formatting/autolinked-references-and-urls. Accessed: 2024-06-03.

[4] 2024. GH Archive: A Public Dataset of GitHub Activity. https://www.gharchive.org/. Accessed: 2024-04-01.

[5] 2024. National Vulnerability Database (NVD). https://nvd.nist.gov/ Accessed: 2024-04-01.

[6] 2024. NVD Data Feed. https://nvd.nist.gov/vuln/data-feeds. Accessed: 2024-04-01.

[7] 2026. *Replication package.* https://zenodo.org/records/17970073

[8] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser. 2022. How Long Do Vulnerabilities Live in the Code? A {Large-Scale} Empirical Measurement Study on {FOSS} Vulnerability Lifetimes. In *31st USENIX Security Symposium (USENIX Security 22)*. 359–376.

[9] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering* 28, 3 (2023), 59.

[10] Hattan Althebeiti and David Mohaisen. 2023. Enriching vulnerability reports through automated and augmented description summarization. In *International Conference on Information Security Applications*. Springer, 213–227.

[11] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.

[12] Leyla Bilge and Tudor Dumitraş. 2012. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 833–844.

[13] Noah Bühlmann and Mohammad Ghafari. 2022. How do developers deal with security issue reports on github?. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. 1580–1589.

[14] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Jianxing Yu, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2025. When chatgpt meets smart contract vulnerability detection: How far are we? *ACM Transactions on Software Engineering and Methodology* 34, 4 (2025), 1–30.

[15] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 654–668.

[16] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of chatgpt model for vulnerability detection. *arXiv preprint arXiv:2304.07232* (2023).

[17] Bharath Chintagunta, Namit Katariya, Xavier Amatriain, and Anitha Kannan. 2021. Medically aware GPT-3 as a data generator for medical dialogue summarization. In *Machine Learning for Healthcare Conference*. PMLR, 354–372.

[18] Shivansh Chopra, Hussain Ahmad, Diksha Goel, and Claudia Szabo. 2024. Chatnvd: Advancing cybersecurity vulnerability assessment with large language models. *arXiv preprint arXiv:2412.04756* (2024).

[19] Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International conference on software maintenance*. IEEE, 516–519.

[20] Roland Croft, M Ali Babar, and M Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 121–133.

[21] Satyaki Das, Syeda Tasnim Fabiha, Saad Shafiq, and Nenad Medvidovic. 2025. Are we learning the right features? a framework for evaluating dl-based software vulnerability detection solutions. *arXiv preprint arXiv:2501.13291* (2025).

[22] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2–12.

[23] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*. 181–191.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[25] Aaron Yi Ding, Gianluca Limon De Jesus, and Marijn Janssen. 2019. Ethical hacking for boosting IoT vulnerability management: A first look into bug bounty programs and responsible disclosure. In *Proceedings of the Eighth International Conference on Telecommunications and Remote Sensing*. 49–55.

[26] GitHub Documentation. 2023. *GitHub REST API*. https://docs.github.com/en/rest Accessed on September 14, 2023.

[27] Etherpad Lite Community. 2024. Security Policy for the Etherpad Lite Repository. Website. https://github.com/ether/etherpad-lite/security Accessed: May 16, 2024.

[28] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[29] Apache Software Foundation. Year. ASF PROJECT SECURITY FOR COMMITTERS. https://www.apache.org/security/committers.html.

[30] Jill J Francis, Marie Johnston, Clare Robertson, Liz Glidewell, Vikki Entwistle, Martin P Eccles, and Jeremy M Grimshaw. 2010. What is an adequate sample size? Operationalising data saturation for theory-based interview studies. *Psychology and health* 25, 10 (2010), 1229–1245.

[31] Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 608–620.

[32] Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. 2023. Chatgpt for vulnerability detection, classification, and repair: How far are we?. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 632–636.

[33] GitHub. 2023. GitHub Security Advisories. Available at https://docs.github.com/en/code-security/security-advisories.

[34] GitHub. Year. About coordinated disclosure of security vulnerabilities. https://shorturl.at/wJO01.

[35] GitHub, Inc. 2024. GitHub REST API v3: Issues Timeline. https://docs.github.com/en/rest/issues/timeline. Accessed: 2024-04-01.

[36] Google. Year. A guide on coordinated vulnerability disclosure for open source projects. includes templates for security policies (security.md) and disclosure notifications. https://github.com/google/oss-vulnerability-guide.

[37] Google. Year. Security techniques - Vulnerability disclosure. https://www.iso.org/standard/72311.html.

[38] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*. 596–607.

Just Accepted

[39] Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. 2019. The review linkage graph for code review analytics: A recovery approach and empirical study. In *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 578–589.
[40] Sameera Horawalavithana, Abhishek Bhattacharjee, Renhao Liu, Nazim Choudhury, Lawrence O. Hall, and Adriana Iamnitchi. 2019. Mentions of security vulnerabilities on reddit, twitter and github. In *IEEE/WIC/ACM International Conference on Web Intelligence*. 200–207.
[41] Allen D Householder, Garret Wassermann, Art Manion, and Chris King. 2017. The cert guide to coordinated vulnerability disclosure. *Software Engineering Institute, Pittsburgh, PA* (2017).
[42] Nasif Imtiaz, Aniqa Khanom, and Laurie Williams. 2022. Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1540–1560.
[43] Javier Luis Cánovas Izquierdo, Valerio Cosentino, Belén Rolandi, Alexandre Bergel, and Jordi Cabot. 2015. GiLA: GitHub label analyzer. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 479–483.
[44] Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. 2021. Predicting issue types on GitHub. *Science of Computer Programming* 205 (2021), 102598.
[45] SHAMBAVI SADAYAPPAN KATHLEEN METRICK, JARED SEMRAU. Year. A guide on coordinated vulnerability disclosure for open source projects. includes templates for security policies (security.md) and disclosure notifications. https://www.mandiant.com/resources/blog/time-between-disclosure-patch-release-and-vulnerability-exploitation.
[46] Madhup Kumar and Ekbal Rashid. 2018. An efficient software development life cycle model for developing software project. *International Journal of Education and Management Engineering* 8, 6 (2018), 59–68.
[47] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2201–2215.
[48] Lisha Li, Zhilei Ren, Xiaochen Li, Weiqin Zou, and He Jiang. 2018. How are issue units linked? empirical study on the linking behavior in github. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 386–395.
[49] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. 2024. On the Effectiveness of Function-Level Vulnerability Detectors for Inter-Procedural Vulnerabilities. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.
[50] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd annual conference on computer security applications*. 201–213.
[51] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258.
[52] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).
[53] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).
[54] Kylie McClanahan, Sky Elder, Marie Louise Uwibambe, Yaling Liu, Rithyka Heng, and Qinghua Li. 2024. When ChatGPT Meets Vulnerability Management: the Good, the Bad, and the Ugly. In *IEEE Int'l Conf. on Computing, Networking and Communications (ICNC)*.
[55] Microsoft. Year. Microsoft's Approach to Coordinated Vulnerability Disclosure. https://www.microsoft.com/en-us/msrc/cvd.
[56] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. 2015. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *2015 IEEE symposium on security and privacy*. IEEE, 692–708.
[57] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Xuan-Bach D Le, and David Lo. 2022. Vulcurator: a vulnerability-fixing commit detector. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1726–1730.
[58] Giang Nguyen-Truong, Hong Jin Kang, David Lo, Abhishek Sharma, Andrew E Santosa, Asankhaya Sharma, and Ming Yi Ang. 2022. Hermes: Using commit-issue linking to detect vulnerability-fixing commits. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 51–62.
[59] OpenAI. 2024. GPT-4o. https://openai.com/index/hello-gpt-4o/. Accessed: 2025-02-01.
[60] Jordan Ott, Abigail Atchison, Paul Harnack, Adrienne Bergh, and Erik Linstead. 2018. A deep learning approach to identifying source code in images and video. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 376–386.
[61] David N Palacio, Daniel McCrystal, Kevin Moran, Carlos Bernal-Cárdenas, Denys Poshyvanyk, and Chris Shenefiel. 2019. Learning to identify security-related issues using convolutional neural networks. In *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 140–144.
[62] Shengyi Pan, Jiayuan Zhou, Filipe Roseiro Cogo, Xin Xia, Lingfeng Bao, Xing Hu, Shanping Li, and Ahmed E Hassan. 2022. Automated unearthing of dangerous issue reports. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 834–846.

Just Accepted

[63] FR Parente, Emanuel B Rodrigues, and César LC Mattos. 2025. FRAPE: A Framework for Risk Assessment, Prioritization and Explainability of vulnerabilities in cybersecurity. *Journal of Information Security and Applications* 89 (2025), 103971.
[64] Van L Parsons. 2014. Stratified sampling. *Wiley StatsRef: Statistics Reference Online* (2014), 1–11.
[65] George Prodan and Elena Pelican. 2023. Prompt scoring system for dialogue summarization using GPT-3. *Authorea Preprints* (2023).
[66] CVE Program. 2023. CVE Services API. Available at https://github.com/CVEProject/cve-services.
[67] Revanth Gangi Reddy, Heba Elfardy, Hou Pong Chan, Kevin Small, and Heng Ji. 2023. Sumren: Summarizing reported speech about events in news. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 12808–12817.
[68] Luis Gustavo Araujo Rodriguez, Julia Selvatici Trazzi, Victor Fossaluza, Rodrigo Campiolo, and Daniel Macêdo Batista. 2018. Analysis of vulnerability disclosure delays from the national vulnerability database. In *Workshop de Segurança Cibernética em Dispositivos Conectados (WSCDC)*. SBC.
[69] Antonino Sabetta and Michele Bezzi. 2018. A practical approach to the automatic classification of security-relevant commits. In *2018 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 579–582.
[70] Clemens Sauerwein, Christian Sillaber, Michael M Huber, Andrea Mussmann, and Ruth Breu. 2018. The tweet advantage: An empirical analysis of 0-day vulnerability information shared on twitter. In *ICT Systems Security and Privacy Protection: 33rd IFIP TC 11 International Conference, SEC 2018*. Springer, 201–215.
[71] Adriana Sejfia, Satyaki Das, Saad Shafiq, and Nenad Medvidović. 2024. Toward Improved Deep Learning-based Vulnerability Detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
[72] Robert W Service. 2009. Book Review: Corbin, J., & Strauss, A.(2008). Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory . Thousand Oaks, CA: Sage. *Organizational Research Methods* 12, 3 (2009), 614–617.
[73] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X Liu. 2012. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 771–781.
[74] Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th ieee/acm international conference on software engineering*. 1–13.
[75] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2237–2248.
[76] Shichao Wang, Yun Zhang, Liagfeng Bao, Xin Xia, and Minghui Wu. 2022. Vcmatch: a ranking-based approach for automatic security patches localization for OSS vulnerabilities. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 589–600.
[77] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. Patchrnn: A deep learning-based system for security patch identification. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*. IEEE, 595–600.
[78] David Wicks. 2017. The coding manual for qualitative researchers. *Qualitative research in organizations and management: an international journal* 12, 2 (2017), 169–170.
[79] Yuki Yoshimura, Shun Shiramatsu, and Takeshi Mizumoto. 2023. Semi-automatic Summarization of Spoken Discourse for Recording Ideas using GPT-3. *IIAI Letters on Informatics and Interdisciplinary Research* 3 (2023).
[80] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2024. Prompt-enhanced software vulnerability detection using chatgpt. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 276–277.
[81] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. 2021. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 705–716.
[82] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).

Just Accepted