Where is it? Tracing the Vulnerability-Relevant Files from Vulnerability Reports

Jiamou Sun
CSIRO
Sydney, NSW, Australia
Frank.Sun@data61.csiro.au

Jieshan Chen
CSIRO
Sydney, NSW, Australia
Jiechan.Chen@data61.csiro.au

Zhenchang Xing
CSIRO & Australian National University
Canberra, ACT, Australia
zhenchang.Xing@anu.edu.au

Qinghua Lu
CSIRO
Sydney, NSW, Australia
Qinghua.Lu@data61.csiro.au

Xiwei Xu
CSIRO
Sydney, NSW, Australia
Xiwei.Xu@data61.csiro.au

Liming Zhu
CSIRO & University of New South Wales
Sydney, NSW, Australia
Liming.Zhu@data61.csiro.au

ABSTRACT

With the widely usage of open-source software, supply-chain-based vulnerability attacks, including SolarWind and Log4Shell, have posed significant risks to software security. Currently, people rely on vulnerability advisory databases or commercial software bill of materials (SBOM) to defend against potential risks. Unfortunately, these datasets do not provide finer-grained file-level vulnerability information, compromising their effectiveness. Previous works have not adequately addressed this issue, and mainstream vulnerability detection methods have their drawbacks that hinder resolving this gap. Driven by the real needs, we propose a framework that can trace the vulnerability-relevant file for each disclosed vulnerability. Our approach uses NVD descriptions with metadata as the inputs, and employs a series of strategies with a LLM model, search engine, heuristic-based text matching method and a deep learning classifier to recommend the most likely vulnerability-relevant file, effectively enhancing the completeness of existing NVD data. Our experiments confirm that the efficiency of the proposed framework, with CodeBERT achieving 0.92 AUC and 0.85 MAP, and our user study proves our approach can help with vulnerability-relevant file detection effectively. To the best of our knowledge, our work is the first one focusing on tracing vulnerability-relevant files, laying the groundwork of building finer-grained vulnerability-aware software bill of materials.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

vulnerability-relevant file, security, software supply chain

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639202

ACM Reference Format:
Jiamou Sun, Jieshan Chen, Zhenchang Xing, Qinghua Lu, Xiwei Xu, and Liming Zhu. 2024. Where is it? Tracing the Vulnerability-Relevant Files from Vulnerability Reports. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639202

1 INTRODUCTION

Software industry heavily relies on Open-Source Software (OSS) not only because of its cost-effectiveness and flexibility but also because it allows developers to provide their own services instead of duplicating existing programming efforts [68]. Statistic reveals that OSS is widely adopted in 1,500 apps across 17 industries, encompassing 98% of the software in overall statistical samples [48]. However, despite its essentiality, vulnerabilities are prevalent among these OSS projects due to inadvertent human errors, posing significant risks to downstream users. Malicious sabotages, such as the infamous SolarWinds [70] and Log4Shell [24] attacks, which exploit weaknesses in the software supply chain, have resulted in losses of billions of dollars. Moreover, the alarming reality is that a single OSS package can have thousands of dependents [41], with a striking 84% of them containing one or more vulnerabilities [68]. As a result, the number of attacks on software dependencies has surged more than sixfold in 2021 [66], making it crucial to prioritize software supply chain security to mitigate potential damages and losses.

Presently, the detection of disclosed vulnerabilities heavily relies on public advisories, acting as dictionaries with unique identifiers and basic characteristics of discovered vulnerabilities. An illustrative example (see Figure 1) is the National Vulnerability Database (NVD), supported by the U.S. government [39], which stands as one of the industry's most crucial and comprehensive vulnerability advisories. Each entry in these databases holds vital information, such as vulnerability's Common Vulnerabilities and Exposures (CVE) identifier, Common Platform Enumeration (CPE) [12], publish date, and features like the vulnerable version, component, and vulnerability type of the affected product. Regrettably, despite their significance, none of these vulnerability advisory databases provide sufficient information on the software supply chain about the affected downstream libraries and products. This lack of supply chain data poses a

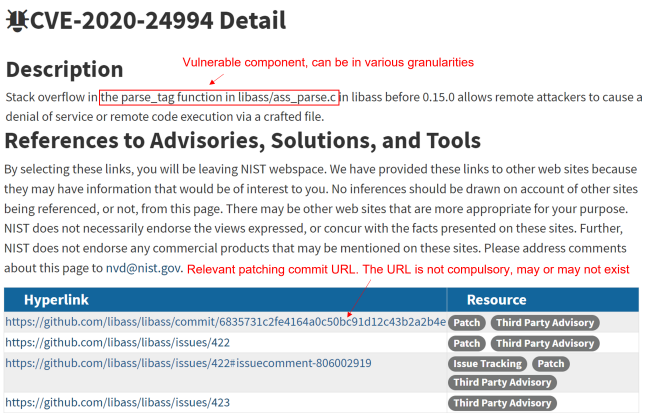


Figure 1: A Screenshot of NVD Entry NVD-2020-24994

significant limitation in comprehensively addressing vulnerabilities and enhancing overall cybersecurity measures.

In response to the critical situation, the concept of a software bill of materials (SBOM) has emerged to provide software package dependencies and dependents, which enhances the transparency of Open-Source Software (OSS) applications. Through SBOM, OSS users can gain valuable insights into the propagation of vulnerabilities, and thus, propose more targeted defense against software supply chain attacks. Various organizations, including Google [42] and OWASP Foundation [44], have noticed the value of SBOM and started establishing their own SBOMs. However, many details regarding SBOM establishment remain undisclosed, with these SBOMs typically having rough-level package dependency relations directly from different package management platforms. These packages exhibit varying complexities, with some comprising only one file (e.g., the NPM package "path-to-regexp" [45]), while others can include more than a hundred files (e.g., the NPM package "server" [54]). Such coarse granularity complicates vulnerability propagation analysis [37], and impairs the SBOMs capacity to pinpoint and mitigate supply chain risks effectively.

Apart from the industrial efforts, some researchers also start working on the vulnerability propagation, but they also face challenges such as coarse granularity, low accuracy, or lack of traceability. For example, some works explicitly trace the library for each disclosed OSS vulnerability from NVD using classification methods [8, 26, 36], but they still focus on package-level vulnerability-relevant library discovery. Other research efforts sought to identify affected files and functions through static analysis or machine learning based on the original vulnerable code [18, 27, 29–31, 46, 55, 77]. Unfortunately, these approaches experienced low accuracies, leading to an abundance of false alerts that overwhelm experts. Moreover, the discovered vulnerabilities lacked traceability to their CVE ids, rendering critical knowledge bases like MITRE ATT&CK [38], CWE [13], and CAPEC [11] inaccessible, which severely compromises their efficacy in safeguarding the software supply chain.

Figure 1 reveals that the NVD's vulnerable component, part of its descriptions, pinpoints elements in software linked to vulnerabilities. Defined in the NVD template [14], it standardizes the disclosure of essential vulnerability information. We found these components useful in identifying specific vulnerability-relevant files. For example, in NVD-2020-24994, *libass/ass_parse.c* clearly indicates the affected file. However, their detail varies from modules to functions. In Figure 1, *parse_tag* refers to a function, while *PostgreSQL adapter* in NVD-2021-22880 indicates a module, and some, like NVD-2022-1222, don't specify components at all. This inconsistency, along with different expressions, renders simple text-matching ineffective for finding related libraries. Recently, Sun et al. [58, 59] used BERT-based Named Entity Recognition (NER) for vulnerability extraction, training models to identify key vulnerability aspects, including vulnerable components in NVD entries. Leveraging their approach, we can precisely identify vulnerable components and use them to predict relevant files.

In this paper, we propose the first work focusing on file-level vulnerability-relevant library tracing, which lays the foundation for constructing finer-grained SBOMs and enabling more accurate software supply chain-based vulnerability detection. Figure 2 provides an overview of our methodology. We first build on Sun et al.'s works [58, 59] by employing a BERT-based model to gather NVD entries with vulnerable components. Next, we employ a GPT-3 model and the GitHub search engine to locate the corresponding GitHub repositories for the affected OSS products, guided by the extracted vulnerable components. Using vulnerability publish dates, we select the commits most likely to contain the vulnerability-relevant files, and obtain a set of potential vulnerability-relevant file candidates. By analysing the file paths, we apply a heuristic-based approach to exclude files that clearly do not represent vulnerability-relevant files. Finally, we train deep learning classifiers to determine whether the file candidates are relevant to the vulnerabilities or not. We adopt the confidence score as the correlation score for ranking the relevancy of each file.

Since no existing study provides a dataset with CVE-vulnerability relevant file pairs, we initially collect CVE-patching commit pairs, and propose heuristic methods to derive the required CVE-relevant file pairs. We finally acquire 81,473 CVE-file candidates corresponding to 2,258 NVD entries as the training and testing dataset for classification, with 3,166 positive data, and 78,307 negative data (Note that we only evaluate on these data, but our approach does not limit to them). We conduct several experiments to evaluate the robustness and effectiveness of our data collection process and classification. The results indicate that the ground truth data collection process is highly robust, achieving an accuracy of 99.7%. Among several baselines, CodeBERT has the best outcomes, achieving an impressive 0.92 in Area Under the Curve (AUC) scores and a Mean Average Precision (MAP) score of 0.85. We further conduct a user study comparing our approach's effectiveness to manual exploration, finding that our method significantly improved the identification of vulnerability-relevant files for disclosed vulnerabilities. This was particularly helpful when dealing with unclear component descriptions. This paper mainly makes the following contributions:

- We contribute the first and largest dataset of 9,435 CVE-vulnerability relevant file pairs linked to 9,021 CVEs, which serves as a valuable resource for future research into the detection of software supply chain risks[1].

[1]https://github.com/anonymous-77400046/vulnerability_file_trace.git

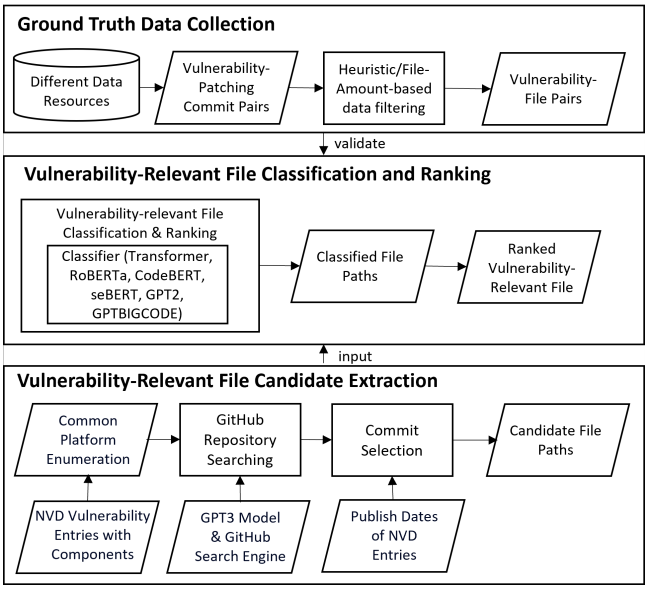


Figure 2: Methodology Overview

Figure 3: A Screenshot of GitHub Commit:2a2b4e corresponding to NVD-2020-24994

- We introduce the first framework to trace vulnerability-relevant components at a file level, enabling granular analysis of security risks in software dependencies and enhancing vulnerability detection and management.
- We undertake a series of experiments, which validate the effectiveness of our approach and the quality of the collected dataset.
- We carry out a user study validating our method in finding vulnerability-relevant files, paving the way for finer-grained, vulnerability-aware SBOM construction.

2 METHODOLOGY

In this section, we present our approach enhancing the traceability of the software supply chain by localising vulnerability-relevant file given a CVE description. As seen in Figure 2, our system consists of two main steps, namely ground truth data collection and the vulnerability-relevant file tracing process. For the ground truth data collection, we initially collected CVE-patching commit pairs from various data sources, and then proposed heuristic methods to transform the collected data into CVE-relevant file pairs. Moving on to the vulnerability-relevant file tracing, we utilized repository collection, GitHub commit collection, and data cleaning procedures. We employed various deep learning models to classify the vulnerability-relevant files among all candidates.

2.1 Ground Truth Data Collection

2.1.1 CVE-Patching Commit Pairs Collection. As mentioned in Section 1, security databases, including NVD and GitHub Advisory Database, do not explicitly mention the vulnerability-relevant file in their records. However, some, though not all (only 0.09% by statistics from our collections), entries will contain the corresponding patching commits for the disclosed vulnerabilities. For instance, the references of NVD-2020-24994 contains the related patch as seen in Figure 1. The patching commit consists of target vulnerability-relevant files along with other associated files. Figure 3 demonstrates the patching commit for NVD-2020-24994. The file *libass/ass_parse.c* is the target vulnerability-relevant file, while the *Changelog* is patch-associated file but not directly relevant to the vulnerability. Based on this observation, we first collected the CVE-patching commit pairs from various sources, and then extracted the related files from changed files in current commit as the ground truth candidates. We collected patching commits from projects including Sun et al.'s work [58], Xu et al.'s work [68], and several security databases, including GitHub Advisory Database [23], Snyk Vulnerability Database [57], OSV Vulnerability Database [62], Debian Security Tracker Database [52], and Security Working Group Node.js Vulnerability Database [53]. Consequently, we acquired 33,999 CVE-patching commit pairs corresponding to 17,293 CVEs.

2.1.2 CVE-Relevant File Pairs Collection. After patch collection, we filtered out files that are irrelevant to vulnerabilities by heuristic-based text-matching methods. As shown in Table 1, we identified 9 file types that are highly likely to be unrelated to the vulnerabilities. Among these nine file types, we compiled 58 postfix/word patterns based on the filename or file path. If a file matches one of these patterns (postfix, file name or file path), we consider it irrelevant and remove it from the vulnerability-relevant candidate list. These patterns will also be instrumental in assisting our classification process in Section 2.2.4. Consequently, we had 104,586 CVE-relevant file pairs corresponding to 16,936 CVEs.

After filtering, we still had many patches with multiple changed files with them. For example, for the patch commit:1c185f of the NVD-2016-9538, two files, *tools/tiffcp.c* and *tools/tiffcrop.c*, were still present after filtering.

The relationship between vulnerabilities and the patched vulnerable files can be complex. A vulnerability can have a single commit with a single changed file as the patch, or it can have multiple patching commits across different branches and repositories with

Table 1: Postfix/Word Patterns for Irrelevant File Filtering

File Types	Postfix/Word Patterns	Scope
Document and text files	.md, .txt, .docx, .pdf, .rst, .changes, .rdoc, .mdown	name
	changelog, news, changes, version, readme, license, authors, todo, history, copying, relnotes, thanks, notice, whatsnew, notes, release_notes	name
	note, license	path
Command and test files	.command, .out, .err, .stderr, .stdout, .test	name
	testlist, testsuite, test	name
	test	path
Images or media files	.jpg, .png, .svg, .mp4, .gif, .exr	name
Data transfer files	.csv, .rdf	name
Font files	.ttf, .otf, .woff, .woff2	name
Mock or stub files	.mock, .stub, .fake	name
Presentation files	.pptx, .key	name
Backup files	.bak, .zip, .gz, .rar	name
GitHub files	.gitignore	name

several different files [68]. Due to the vast amount of data and the complexity between vulnerability and the true patching files, it becomes challenging to determine whether these files are indeed associated with vulnerabilities. To mitigate the impact of the false positive files, we chose to further remove the CVE-patching commit pairs with multiple involved files. This resulted in retention of only those CVE-patching commit pairs that had a clear one-to-one relationship with CVE-relevant files. While this approach resulted in some data loss, it ensures the accuracy of the overall ground truths. As a result, we had 9,435 CVE-file pairs corresponding to 9,021 CVEs as our final collection of CVE-relevant file pairs. Note that the number of CVE-file pairs are larger than the number of CVEs because some CVEs have the same vulnerability-relevant files in multiple forked repositories (see Section 2.2.2 for details). We keep such information to reserve more ground truths. Table 2a shows the statistics during the ground truth collection steps.

2.2 Vulnerability-Relevant File Candidate Extraction

2.2.1 NVD Collection. As proof of the concept, we conducted our approach to the disclosed NVD vulnerability entries that had explicit textual vulnerable components. All the NVD entries can be downloaded from the official website[2]. In light of the dynamic nature of vulnerability information, where maintainers continuously update details, leading to a time lag in NVD description stabilization [59], we took measures to ensure the quality of NVD entries. Specifically, we restricted our downloads to NVD entries before

[2]https://nvd.nist.gov/vuln/data-feeds

Table 2: Statistics of Data Collection

(a) Statistics of Ground Truth Collection

	Data Pairs	CVEs
CVE-Patching Commit Pairs	17,293	33,999
CVE-File Pairs with relevant file type	104,586	16,936
CVE-File Pairs with single changed file	9,435	9,021

(b) Statistics of Candidate Collection

	Data Pairs	CVEs
Repository via GPT3	1,060	3,713
Repository via GPT3 & Search Engine	2,421	4,178
CVE-Relevant Commit Candidates	8,097	4,178
CVE-Relevant File Candidates	12,689,667	2,258
CVE-Relevant File Candidates in Training	81,473	2,258

July 2022. We then utilized Sun et al.'s works [58, 59], using BERT-based Name Entity Recognition model to extract the vulnerable components from the NVD descriptions.

Accordingly, we gained 177,718 NVD entries from the NVD database. Since not all NVD descriptions contain vulnerable components (i.e., NVD-2022-1222), after filtering, we only have 68,565 left NVD entries. Since our focus is on NVD entries with a confirmed ground truth of vulnerability-relevant files, we excluded NVD entries that were not presented in our ground truth dataset. Overall, we had 4,438 NVD entries as our targets.

2.2.2 Repository Collection. To begin the process, our first step is to collect OSS repositories associated with the vulnerability entries. A problem is that the repository of the vulnerable product is not the mandatory information in the NVD. Some entries, including NVD-2020-24994 from Figure 1 will record the patching information including the relevant repository links in the reference list, while for most of the entries, such information is not available (e.g., the NVD-2021-22880) [68]. To solve this problem, we leveraged the power of LLM with GitHub search engine together.

As LLM, such as GPT-3 [5], is trained under great amount of online materials, it can remember various information about OSS products including their published online repositories in the GitHub. Hence, for each collected NVD entry, we proposed the following prompt as the input of GPT-3 to obtain the repository:

Give me the GitHub URL of <cpe_product>, answer within 100 tokens.

where the *<cpe_product>* is the product name presenting in the form of Common Platform Enumeration (CPE) [12] that can be directly acquired from the metadata of downloaded NVD dataset. As a consequence, we fetched 1,583 repository URLs corresponding to 4,438 NVD entries, with 1,060 valid repository URLs (i.e., able to be fetched online) corresponding to 3,713 NVD entries. Note that one repository may correspond to multiple NVD entries.

One drawback of GPT-3 is that the training data are all before 2021 [5]. The repositories may undergo changes, including hiding or deletion, by their maintainers afterwards. For example, NVD-2018-17785 includes the GitHub repository blynkkk/blynk-server of the product Blynk, but this GitHub repository is no longer available. Therefore, the GPT-3 results can be inaccessible. Even worse, GPT-3

can create non-existed inaccessible URLs. Fortunately, other developers may fork the repository which contains the vulnerability-relevant file, and thus, we can still get the copy of them even if the original repository is changed into private, deleted, or becomes inaccessible. Therefore, to overcome such issue, we consider GitHub search engine as a supplement to GPT-3 for repository searching. We firstly requested the repository information from the GPT-returned results via GitHub REST APIs[3]. If the URL is inaccessible or the repository is empty, we retrieve the possible forked repositories by GitHub search engine. Again, we used product names from CPE as the searching keywords, and kept the top-5 returns as the results. Through the combination of GPT-3 and GitHub search engine, we eventually procured 2,421 repository candidates corresponding to 4,178 NVD entries. The missing of the crawled NVD entries is due to the unavailability of the repository results from the combination of GPT-3 and GitHub search engine.

2.2.3 Commit Collection. After the repository collection, we started to compile the possible commits containing vulnerability-relevant files. The structure and file trees of the OSS product can be changed by the maintainers overtime, so the original vulnerability-relevant files can be combined with other files, moved to other locations, or deleted. Thus, it is ideal to acquire the commit that is closer to the time of vulnerability patching. Since the patching commit is unavailable for most of the NVD entries as mentioned in Section 2.1.1, we employed the publish date of NVD entry as the timestamp and tried to find the closest commit earlier. The publish dates of NVD entries can be directly acquired from the metadata of the official downloaded dataset. Nevertheless, the closest commit can only be requested by GitHub REST APIs, which have the rate limit of 5,000 requests per hour. Considering that OSS projects like Linux [34] can have thousands of commits during a short period of time, parsing all historical commits and selecting the most suitable one would be impractical. To manage this limitation, we set a threshold of 90 days to limit the overall requests. If we could not find any commits within 90 days prior to the NVD publish date, we would terminate the searching process and instead utilize the default commit, i.e. the newest commit, as the target. By this approach, we eventually collected 8,097 commits involving 28,385,423 files across 1,952 repositories corresponding to 4,178 NVD entries. The missing of the NVD entries is because some repositories are empty so no commit can be crawled.

2.2.4 Vulnerability-Relevant File Candidates Collection. As mentioned before, the description of vulnerable component of NVD can be vague and in various levels of granularity. In particular, the vulnerable component illustrated in Figure 1 is in function level, while the description of NVD-2021-22880 presents the module-level component. This discrepancy requires us to fetch the finer-grained file information to assist the deep learning models trace the real vulnerability-relevant files. We used GitHub REST APIs to request all the contents of files from collected commits. Similarly, we used proposed heuristic-based rules in Section 2.1.2 and Table 1 to filter the irrelevant files out and parse all the code contents from the remaining files. However, due to GitHub REST API has limitation of 5,000 requests per hour (see Section 2.2.3), we were only able to collect a limited amount of contents for the involved files. This process took us over two thousand hours (more than three months) to complete. As a result, we acquired 12,689,667 file contents across 848 repositories corresponding to 2,258 NVD entries. Table 2b shows the statistics of collected data during the candidate collection steps.

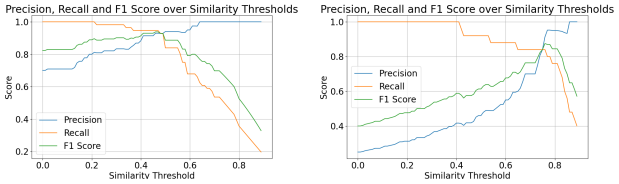
[3]https://docs.github.com/en/rest?apiVersion=2022-11-28

2.3 Vulnerability-Relevant File Classification and Ranking

2.3.1 File Candidate Matching. We used the ground truths in the form of CVE-relevant file pairs collected in Section 2.1.2 to construct the training set by refining Vulnerability-Relevant File Candidates dataset collected in Section 2.2.4. For each conducted vulnerability-relevant file candidate, if it has the same repository, file path, and file name to the ground truth, we think the vulnerability-relevant file candidate is the actual vulnerability-relevant file of the vulnerability.

For the rest files, we further adopted a threshold-based method to measure whether they are correct (positive) pairs or incorrect (negative) pairs. As mentioned in Section 2.2.2, a vulnerability can have multiple patching commits due to repository forking. This leads to the discrepancy of vulnerability-relevant file name, which means the vulnerability-relevant file for the same vulnerability can be scattered in different repositories with different paths. Simultaneously, one product can adopt components from another product with few changes. For example, the GitHub repository bonzini/qemu make use of the file */hw/scsi-disk.c* that directly from another file */hw/scsi/scsi-disk.c* from the repository qemu/qemu. From the ground truth, the file */hw/scsi-disk.c* is the vulnerability-relevant file for product QEMU of NVD-2011-3346. In such case, the file */hw/scsi-disk.c* and file */hw/scsi/scsi-disk.c* should be considered as the same file, and the file */hw/scsi/scsi-disk.c* can also be seen as the vulnerability-relevant file of NVD-2011-3346. This is because once the file */hw/scsi/scsi-disk.c* is confirmed to have the similar code content to the real vulnerability-relevant file, then it will be easier for security experts to find the real vulnerability-relevant file in the true repository of the vulnerable product. To solve such problem, for each file candidate of each vulnerability, if it has the same file name with the vulnerability-relevant file in ground truth, we calculated the content similarity between the two files. We embedded the file contents into TF-IDF vectors and calculated the cosine similarity [50, 51]. We consider the two files are the same if their similarity score is above a certain threshold.

To determine the threshold, we categories them into two groups, (the first group) where the file path in the forked repository remains the same as the original, implying fewer changes, and (the second group) where the file path has been altered, suggesting more substantial modifications. For each group, to ensure our sampled dataset accurately reflects the diversity of differences, we stratify the sampling across intervals of similarity scores. Specifically, we select 10 file pairs from each decile range of similarity scores, such as 10 file pairs from a 0-0.1 similarity score range, another 10 from a 0.1-0.2 range, and so on. This stratification helps to maintain the representativeness of the sampled dataset across the spectrum of potential modifications. Since not every interval contains at least



(a) Scores of File Pairs with Same File Path but Different Repositories

(b) Scores of File Pairs with Different File Paths and Repositories

Figure 4: Precision, Recall, and F1 scores under Different Thresholds

10 samples, we opted to include all available samples in the intervals that do not meet this criterion. Consequently, for the first and second groups, we obtained 80 and 100 samples, respectively. Two annotators separately labelled the data and compared, and if the contents between the matching file pairs are same, or the matching file contains functions recorded as vulnerability-relevant, we consider the matching file is a near-duplicate/copy of the recorded vulnerability-relevant file. If there was any disagreement, the two authors would discuss and get the consensus. We adopted Cohen's Kappa [10] to validate the agreement level. The Cohen's Kappa between two authors was 0.86, which indicated an almost perfect agreement. This was because we only kept the patching commit with single vulnerability-relevant file, so the vulnerability-relevant function was clear. After labelling, we tested various thresholds to evaluate their impact on precision (TP/(TP+FP)), recall (TP/(TP/FN)), and F1 score (2*Precision*Recall/(Precision+Recall)). The objective was to find the ideal balance that maximizes the inclusion of true positive pairs (genuine vulnerable files) and minimizes the presence of false positives (irrelevant files), thereby refining the dataset for classifier training. A matching file is considered as a true positive (TP) if its similarity score exceeds the set threshold and it is or is a near-duplicate/copy of the groundtruth file. Conversely, a matching file is labeled as a false positive (FP) if the similarity score is larger than the threshold but it is not a copy/near duplicate of the groundtruth file. Figure 4a and Figure 4b show the final results, respectively. From the figure, we eventually selected 0.486 and 0.75 as the thresholds for the first and second groups respectively. Once we obtained these thresholds, we apply them to further refine our dataset obtained in Section 2.2.4 by removing the pairs when the similarity score between the retrieved file and the ground truth file is less than the thresholds.

2.3.2 Vulnerability-Relevant File Classification. We used a binary classifier to determine whether a file candidate can be recognized as the vulnerability-relevant file of NVD entry. In this work, we experimented with several classification models:

Transformer-based: The Transformer model [60] utilizes bidirectional self-attention techniques, which sets it apart from traditional NLP models, and has demonstrated remarkable effectiveness in various NLP tasks [60]. Many of the currently popular models, such as BERT [15] and GPT-2 [47], are built on the foundations of the Transformer architecture. We use vanilla **Transformer** as the representation of non-pre-trained small language models. The SoftMax layer is added at the end of the Transformer encoders to form the binary classifier.

BERT-based: Following the Transformer architecture, BERT was introduced by stacking multiple Transformer encoders. In comparison to the original Transformer, BERT features more extensive and deeper neural network structures and has undergone pre-training with substantial amounts of data, using techniques like masked LM and sentence prediction. Through fine-tuning, BERT has demonstrated remarkable advantages in tackling traditional NLP tasks [15]. In this paper, we use **RoBERTa** [35], **CodeBERT** [20], and **seBERT** [61] models as baselines of pre-trained language model. RoBERTa differs from BERT in that it was trained on a larger dataset, and incorporates more effective pre-training procedures. On the other hand, CodeBERT was trained using both semantic and code-related information, making it well-suited for code-related tasks. seBERT was purely trained on software semantic data.

LLM-based: Leveraging the advancements in computing power, the language model becomes unprecedented complex and powerful. LLMs, with exponential growing sizes and tremendous training data, have become one of the most popular semantic models. While LLMs, including GPT models, are also built upon the Transformer architectures, they benefit from larger and more comprehensive training datasets leading to remarkable capabilities in handling diverse NLP tasks [47]. Due to the limitation of hardware, we use **GPT-2** [47] and **GPTBigCode** [1] as LLM-based baselines, to show the capability of in tracing the vulnerability-relevant files. GPTBigCode, built on GPT-2, is pre-trained specifically using code contents, making it potentially more suitable for our task.

The input to all classifiers is the concatenation of the vulnerable component described in the NVD description, the name and path of the crawled repository and file candidates, and the file contents. The output is the binary classifications with confidence score indicating whether the file candidate is related to the vulnerability or not. Note that from Sun et al.'s work [59], the BERT model cannot perform perfectly in name entity recognition (NER) tasks, leading to potential false positives or false negatives in identifying vulnerable components. To address this concern, we employ a strategy where we use the first 80 words in the NVD description as the training input, instead of directly extracting vulnerable components for training. The rationale behind choosing 80 words is that we observed that the description of the vulnerable component typically appears at the beginning of the overall description (as seen in Figure 1). This approach helps avoid potential inaccuracies in NER and improves the overall performance of the models. Furthermore, as Transformer-based models have input length limitations, we further restricted the input to 512 tokens per model by using the first 80 words from the NVD description as the vulnerable component description and pruning any overlength file contents. This ensures that the input size remains within the models' capacity. We employ 128 input length to seBERT as it was pre-trained under this length.

2.3.3 Vulnerability-Relevant File Ranking. For the classification results of a vulnerability, we directly use the confidence scores of the positive classification as correlation scores to rank the relevancy of candidates. The vulnerability candidate with the highest score will be recommended at the top.

3 EXPERIMENTS

In this section, we conducted experiments to validate the quality of our vulnerability-relevant file tracing method. To evaluate the performance of our approach, we propose the following five research questions (RQs):

- What is the accuracy of constructed ground truth of CVE-relevant file pairs?
- What is the accuracy of patterns for irrelevant file filtering?
- What is the accuracy of collected training and testing sets of vulnerability-relevant file candidates?
- How is the effectiveness of the classifiers?
- How is the usefulness of the proposed approach?

3.1 Accuracy of Constructed Ground Truth

In this section, we evaluated the accuracy of the collected the ground truth data in Section 2.1, i.e., whether the collected file is the real vulnerability-relevant file of the corresponding vulnerability. We followed the statistical method proposed in [56] to sample a subset of the ground truth data and manually validated them. From work [56], the number of samples is calculated by the formula below:

$$MIN = n_0/(1 + (n_0 - 1)/populationsize) \quad (1)$$

$$n_0 = (Z^2 * 0.25)/e^2 \quad (2)$$

where Z represents the z-score corresponding to the confidence level, e signifies the desired error margin, $populationsize$ stands for the total number of data points in the dataset, and MIN Denotes the minimum number of samples required to maintain the error margin e. Here we adopted $e = 0.05$. From Section 2.1.2, we totally collected 9,435 CVE-relevant file pairs. Hence, the population size is 9,435, and the overall sample size is 369.

In consequence, the accuracy of the ground truth is 99.7%. This high accuracy is mainly due to the fact that the correctness of the CVE-vulnerability relevant file pairs depends on the accuracy of the collected CVE-patching commit pairs. Since most of these commit pairs have already been verified by previous research, the obtained result is considered reasonable. The only one inaccurate ground truth found is the CVE-files pair of NVD-2016-1583, where the collected patch commit:b21cd9 mistakenly shows the vulnerability-relevant file is *fs/proc/root.c* but the true vulnerability-relevant file is *fs/ecryptfs/kthread.c* according to the NVD description.

> Overall, the accuracy of our collected CVE-relevant file pairs is as high as 99.7%. The sole mistake is because the CVE-patching commit pair is wrong.

3.2 Accuracy of Irrelevant File Filtering Patterns

In Section 2.1.2, we proposed patterns to filtering irrelevant files. To validate the accuracy, we followed statistical sampling method in Section 3.1. We totally checked 384 sampled results, and we figure out the filtering results can reach 100% accuracy as the filtered files are commonly testing files, images and texts.

> Overall, the accuracy of our proposed irrelevant file filtering patterns is perfect.

3.3 Accuracy of Collected Vulnerability-Relevant File Candidates

3.3.1 Accuracy of File Candidate Matching. In Section 2.3.1, we proposed thresholds to prepare the dataset for the classifier. We calculate the similarity between the groundtruth file and the vulnerability-relevant files we collected in Section 2.2, and use a threshold based method to determine the positive and negative pairs. To validate the quality of the threshold method, we sampled a statistical significant subset of the data using the same statistical sampling method mentioned in Section 3.1. Two annotators separately checked the matching results by comparing the contents of the matched file and the recorded ground truth file. The Cohen's Kappa between them was 0.77, indicating a substantial agreement. After labeling, the two annotators discussed discrepancies and reached a consensus.

As a result, we manually sampled and validated 211 pairs of files with the same file path but in different repositories, and 194 pairs of files with the distinct file paths and repositories. The accuracies are 99.5% and 76.3%, respectively. For the file pairs with only different repositories, the only found mistake is the matching of vulnerability-relevant file *net/sched/act_police.c* from repository aeroevan/android_kernel_asus_grouper corresponding to NVD-2010-3477. The matched file is *net/sched/act_police.c* from repository torvalds/linux. Although the matched file name and repository are correct, due to wrong selection of the commit, the matched file does not contain the vulnerability-relevant function *tcf_act_police_dump*. For the file pairs with total different paths and repositories, there are higher rates of mistakes. As both the repositories and the paths are different, the uncertainties become higher, with more possibilities for vulnerable functions to be deleted, or different files with the same names. Accordingly, the vulnerability-relevant function does not exist in the matched file, even if the matched file name and path are correct.

3.3.2 Accuracy of Collected Classification Dataset. In Section 2.1, we presented heuristic methods for locating the repository and commit related to the disclosed NVD entry. Although these steps are generally reliable, there is still a possibility of encountering mistakes, leading to the chance of locating the wrong repository/commit and facing difficulties in finding the vulnerability-relevant file. To gauge the impact of such errors, we conducted validation to determine whether the collected file candidates for each CVE contain the true positive vulnerability-relevant file. The presence of the true positive file indicates the effectiveness of our collection method. As a result, out of 2,258 CVEs, we successfully located vulnerability-relevant files for 2,029 CVEs, yielding a success rate of 89.9%. The absence of vulnerability-relevant files can be attributed to errors in repository retrieval and incorrect selection of commits. This result shows that our proposed framework can trace the vulnerability relevant files for up to 89.9% CVEs.

> Our file candidate matching can achieve 99.5% and 76.3% accuracy for the file pairs with only different repositories, and the file pairs with different paths and repositories, respectively. Our classifier dataset collection approach can attain true vulnerability-relevant file candidates in 89.9% CVEs. The mistakes are due to incorrect repository retrieval and incorrect selection of commits.

Table 3: Performance of Classifiers

	CodeBERT	RoBERTa	GPT-2	GPTBig Code	Trans -former	seBERT
Pre	**0.56**	**0.56**	0.54	0.51	0.51	0.55
Rec	**0.85**	0.83	0.82	0.75	0.66	0.75
F1	**0.67**	**0.67**	0.65	0.61	0.33	0.64
AUC	**0.92**	0.89	0.88	0.82	0.67	0.79

3.4 Effectiveness of Vulnerability-Relevant File Tracing

3.4.1 Effectiveness of Classifiers. We verify the effectiveness of different classification models described in the Section 2.3.2 in this RQ. Given that different models possess distinct advantages - for instance, BERT-based models are more resource-efficient during runtime due to their smaller size, while GPT-based models offer higher computational power - aims to determine the optimal solutions. All models were implemented using HuggingFace [65] libraries. We used default hyperparameters for all the training. All experiments were done on an Intel Core i9-9900K CPU and a Nvidia Titan V GPU.

The dataset included 81,473 CVE-file candidates corresponding to 2,258 NVD entries, with 3,166 positive CVE-relevant file pairs and 78,307 negative CVE-relevant file pairs. We performed 5-fold cross-validation to verify the effectiveness. To prove the training effect, we used down-sampling methods [7], randomly keeping same amount of positive and negative samples in training set. For the testing set, as we had substantial data, the validation became difficult and time-consuming. Thus, as a substitute, we randomly down-sampled the number of negatives samples to 100 times that of positive samples in the testing set. In this way, we reduced the overall validation time to the 600 GPU hours while keeping the extreme unbalanced data distribution. Because of the data unbalance, we used macro-precision (Pre, the average of precision scores for each class in the classification), macro-recall (Rec, the average of recall scores for each class in the classification), F1 score of macro-precision and macro-recall (F1) [9], and Area Under The Curve (AUC) [25] to measure the training results.

Table 3 shows the performance of different classification models. Surprisingly, CodeBERT performs the best (0.92 AUC), followed by RoBERTa (0.89 AUC), GPT-based models (0.88 and 0.82 AUC for GPT-2 and GPTBigCode, respectively), seBERT (0.79 AUC), and the vanilla Transformer (0.67 AUC). The F1 score results exhibit comparable trends. Both CodeBERT and RoBERTa achieve an F1 score of 0.67, though CodeBERT has a marginally higher recall at 0.85. GPT-2 registers an F1 score of 0.65, trailed by seBERT at 0.64, GPTBigCode at 0.61, and Transformer at 0.33.

While GPT-based models (GPT-2 and GPTBigCode) exhibit effective feature learning attributed to their architectural complexities and capacities, the quality of training data plays a more crucial role in determining their overall performance. By statistics, BERT-based models is pre-trained on BookCorpus [76] and English Wikipedia [19] with total of 3,300 million of words [22]. GPT-2 is pre-train under WebText with 8 million documents that has not been publicly available [22, 47]; however, the authors release part of the dataset [4], of which we can make use to estimate the overall data size. From the released dataset, the 250k documents contain around 124 million words. In a similar fashion, the total WebText can involve around 4 billion words, whose amount of dataset does not exceed the pre-training set of BERT by much. Even worse, the large size of the GPT-based model requires more training set to bring the superiority, which is not align with our situation. Meanwhile, RoBERTa-based model has 10 times of the training set than BERT with more optimal training methods [35], so in our experiments, CodeBERT and RoBERTa can achieve better results.

Table 4: Performance of Ranking

	CodeBERT	RoBERTa	GPT-2	GPTBig Code	Trans -former	seBERT
MAP	**0.85**	0.84	0.27	0.22	0.06	0.22

Compared with models trained on general data, the models that are pre-trained under code contents do not acquire huge advantages. The CodeBERT model is only slightly better than the RoBERTa, while the GPTBigCode is worse than the GPT-2. We believe this is due to the complexity of the vulnerability-relevant codes that contain abundant types of languages. The CodeBERT is only pre-trained under 6 kind of languages including Python, Javascript, Ruby, Go, Java, and PHP code, and the GPTBigCode is only pre-trained under Python, Javascript and Java [1, 20]. Lots of vulnerability-relevant code languages, including C, are ignored, so the code-aware language models are not necessarily better than the base models.

The seBERT model, trained solely on textual data without code information, is limited in comprehending complex vulnerable code patterns. Additionally, the model's training neglects specialized platforms like Information Security Stack Exchange, which could enhance its understanding of vulnerability knowledge. The experimental results show that while seBERT is adept at recognizing software engineering terms like "bug" and "ant" [61], this capability doesn't significantly aid in identifying vulnerable code. Nevertheless, as we imagined, the pre-trained language models with bigger model sizes are much better than the raw Transformer.

By observation, the incorrect classification is because of the following reasons: (1) the model cannot distinguish the similar file names, i.e., the vulnerability-relevant file *SamlHeaderInHandler.java* of NVD-2014-3584 is textual similar to the file candidate *SchemaHandler.java*, so that model cannot differentiate their differences; (2) the vulnerable component extracted in NVD description is false positive, so the model cannot decide the true vulnerability-relevant files, i.e., the false positive extraction *Inf loop* in NVD-2022-1222; (3) There are contradictions between NVD descriptions and the ground truth, i.e., in the NVD-2022-1286, the vulnerable component is described as *mrb_vm_exec*, while the patch commit:3b3ee9 under the reference list shows the vulnerable component is *mrb_remove_method*, confusing the model.

3.4.2 Effectiveness of Ranking. We used confidence scores to rank the vulnerability-relevant file for each NVD. Table 4 shows the results. Correspondingly, the CodeBERT gets the best result (0.85

[4] https://github.com/openai/gpt-2-output-dataset

MAP), followed by RoBERTa (0.84 MAP), GPT-2 (0.27 MAP), seBERT (0.22 MAP), GPTBigCode (0.22 MAP) and Transformer (0.06 MAP). In summary, our CodeBERT model achieves an impressive MAP score of 0.85, demonstrating that our approach is highly effective in addressing the problem of vulnerability-relevant file tracing.

The BERT-based models generally have better results than the GPT-based models, and CodeBERT gets the best outcomes. The pre-trained models are far better than the raw Transformer.

3.5 Usefulness of Vulnerability-Relevant File Traceability

To evaluate the usefulness of our approach, we conducted a user study, comparing the performance of users who utilized our system with those who solely relied on search engines and their own experience.

3.5.1 User Study Design. Participants: We recruited 16 students with IT and security background as the participants. Within them, we have 9 males and 7 females, each with at least four-year experience in programming developing and testing and have basic knowledge of vulnerability analysis as well as vulnerability-relevant library tracing. All the participants are in age of 18-30. We randomly separated them into two groups, i.e., control group and the experimental group. The control group used the online search engine to search for the related vulnerability-relevant file given a NVD entry, while the experimental group could see the top-one vulnerability-relevant file and repository retrieved by our system while searching.

Procedure: The user study was conducted through online questionnaires, with an expected duration of approximately one hour. The participants were provided with a 10-minute tutorial video that explained the study's purpose, their tasks, and the expected time commitment. Two practice tasks were included to familiarize participants with the user study interface and to address any potential misunderstandings. Upon completion of the practice tasks, participants began the formal user study, which consisted of 9 tasks. Each task involved locating vulnerability relevant file based on the provided NVD entry. For the experimental group, we additionally presented the top result from our tool to aid them. Participants were given a 10-minute time limit to complete each question, and the time spent on each question was recorded. Once the time limit elapsed, participants could no longer edit their answers. For the experiment group, after providing their answers, we further asked them to rate the usefulness of our tool using a 5-point Likert scale. The rating time were not included in the time spent for each question.

Task Selection: We categorized the tasks into three difficulty levels: EASY, MEDIUM, and HARD, based on the ease of finding the vulnerability-relevant file given the provided NVD entries. An EASY task is when the path of the vulnerability-relevant file is directly presented in the NVD description or when the relevant patching commit is listed in the NVD reference list. A MEDIUM task is when the NVD description mentions the name of the vulnerability-relevant function, class, or file without providing the exact path of the vulnerability-relevant file. A HARD task is when the NVD description only states the general component or module name without specifying the vulnerability-relevant file's path. We selected a total of 9 tasks related to 9 NVD entries, with 3 tasks representing each difficulty level, for the participants to complete. As our model is not perfect and false positives could potentially mislead users, we intentionally included a false positive result for each difficulty level. This would enable us to assess the impact of incorrect classifications on participants' decision-making process. We listed all the tasks in Table 5 detailing NVD entries, GT vulnerability-relevant file names, top-one candidate and task difficulties, and we add $*$ to the top right corner of top-ranked candidate if they are false positives.

3.5.2 User Study Results. Table 5 presents the average completion time (AvgTime) and correctness (Correct) for each task in both the experimental and control groups. Compared to the control group without any support, the experimental groups, utilizing our approach to retrieve the top-ranked vulnerability-relevant candidate, complete the tasks more efficiently (174.0±71.4 versus 256.0±101.6) and achieve a higher correctness (61/72 versus 39/72). While the experimental group achieve faster completion times and higher correctness for most tasks, there are slight differences in tasks T4 and T7. In these cases, our tool provide false positives to the participants, requiring them to spend additional time verifying the accuracy of the links before following the same procedure as the control group to search for the vulnerability-relevant files.

We conduct a further evaluation to assess the impacts of our model on tasks of varying difficulties. The results indicate that the experimental group outperformed the control group in both completion time and correctness across all difficulty levels. Interestingly, we observe that the impact of our tool increased in terms of correctness from EASY tasks to HARD tasks, with similar correctness scores in EASY (24 versus 19) and MEDIUM (22 versus 19) tasks but a substantial difference in HARD tasks (15 versus 1). On the other hand, the impacts on completion time do not exhibit significant differences across difficulty levels, with completion times of 113.3±11.9 (197.3±61.1) for EASY tasks, 147.3±38.8 (226.2±51.2) for MEDIUM tasks, and 261.5±39.9 (344.5±111.6) for HARD tasks in the experimental (control) groups, respectively. The reason may because participants required more time to validate the top-ranked candidates as the tasks became harder. To understand the impacts of our models in different task difficulties, we adopted the statistical sampling method mentioned in Section 3.1 to sample a significant subset of CVD entries. Among 382 sampled tasks, 32.2% (123/382) are EAST tasks, 29.1% (111/382) are MEDIUM tasks, and the rest 38.7% (148/382) are HARD tasks. These distributions confirm the practicability of our methods. Overall, our model demonstrate notable advantages in improving correctness, particularly for more challenging tasks, while maintaining relatively consistent completion times across different difficulty levels. The distribution of tasks in different difficulties further strengthen the practicability and usefulness of our work.

In relation to the false positives, participants in the experimental groups take extra time to validate the answers compared to the control group. Upon examining all the answers from the experimental groups, we observe that all participants recognized that the provided candidates were false positives. Therefore, despite lower correctness in the experimental group for T4, the reason appears to be attributed to the participants themselves when they tried to find CVE-relevant files using the control group method rather than the models used. Overall, our findings indicate that participants

Table 5: User Study Tasks and Results. * indicates the Top-Ranked Candidate is a false positive. Each task is completed by eight participants in each group.

Index	NVD Entries	Vulnerability-Relevant File	Top-Ranked Candidate	Difficulty	Experimental Group		Control Group	
					AvgTime (s)	#Correct	AvgTime (s)	#Correct
T1	NVD-2015-3308	x509_ext.c	verify.c*	EASY	120.5	8	181.4	8
T2	NVD-2017-13024	print-mobility.c	print-mobility.c	EASY	96.5	8	131.8	6
T3	NVD-2006-6333	ibmtr.c	ibmtr.c	EASY	122.8	8	278.8	5
T4	NVD-2015-4050	fragmentlistener.php	ProfilerListener.php*	MEDIUM	201.0	6	161.8	8
T5	NVD-2017-12867	timelimitedtoken.php	TimeLimitedToken.php	MEDIUM	110.4	8	229.8	7
T6	NVD-2005-3356	mqueue.c	mqueue.c	MEDIUM	130.5	8	287.0	4
T7	NVD-2021-22880	money.rb	connection_handler.rb*	HARD	207.9	2	194.0	0
T8	NVD-2017-10379	mysql.cc	mysql.cc	HARD	272.9	6	460.9	1
T9	NVD-2021-21841	box_code_base.c	box_code_base.c	HARD	303.6	7	378.5	0
				Avg ± Stdev	174.0±71.4		256.0±101.6	

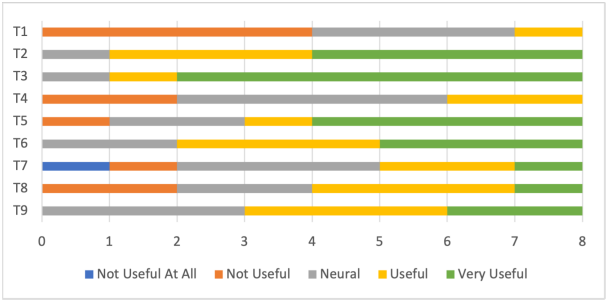


Figure 5: Usefulness Rates. We have 8 ratings for each task.

were able to accurately judge whether the provided assistance was useful or not, and they were indeed aided by our tool.

Figure 5 displays the usefulness ratings for each task in the experimental group. The results suggest that participants in the experimental groups generally perceived our tool as beneficial in identifying vulnerability-relevant files. However, when we compare the tasks with true positives to those with false positives, we observe that tasks associated with false positives received more negative or neutral usefulness ratings. This observation is reasonable as false positives may instead require additional efforts from the participants and they seemed less convinced of its effectiveness when dealing with false positives.

Participants using our tool show faster completion times and higher correctness rates compared to the control group. The tool's impact on correctness is especially evident as the tasks become more challenging. Interestingly, false negatives don't significantly influence participants' judgments, but they do require additional time for validation.

4 DISCUSSION

Our study aims to identify file-level vulnerabilities in disclosed CVEs, providing a more precise approach compared to traditional package-level detection. Previous research [72] shows that despite numerous outdated dependencies, 73.3% of surveyed software doesn't use vulnerable functions, thus avoiding threats. By focusing on file-level details, our method improves detection accuracy and minimizes unneeded updates, saving time and resources.

Our study focuses on CVEs with fixes confined to one file, providing a foundational basis for wider vulnerability analysis. Using feature location techniques (FLT) [16] as a model, our findings serve as a starting point. By integrating static and dynamic code analysis, we can uncover related functions and risk patterns, simplifying comprehensive risk assessment.

Thousands of conducted datasets open the door to new ways of vulnerability detection and management. Utilizing FLT, these datasets act as starting points for detecting vulnerabilities and set the stage for future research into file-level or finer vulnerability identification. Essentially, our work fills current gaps and guides future explorations in this field.

5 THREATS TO VALIDITY

Internal validity: Our ground truth is imperfect, with vulnerabilities often linked to multiple repositories and files, and patching commits that are incomplete or contradictory. Collecting complete ground truth is challenging due to the evolving nature of vulnerabilities and open-source software. We've opted for a one-by-one CVE-relevant file pair approach for ground truth, but this doesn't restrict our model to single-file CVEs. Our work is an initial step in this domain, with the issue of ground truth left for future research.

We set a 90-day threshold for commit parsing to balance time efficiency and effectiveness. While validating every possible threshold is challenging, statistics [33] show that CVE Numbering Authorities, typically technology giants, often fix vulnerabilities within 30 days before to 90 days after disclosure, making a 90-day threshold a reasonable choice.

Due to GitHub's request limitations, we collected only part of the file content, consuming over 2,000 hours. Our approach can process 20 CVEs per day, faster than manual analysis. For industrial application, concurrency techniques could enhance efficiency.

We limited negative samples to 100 times the positive ones due to computational limits, which might impact fairness. Despite data reduction, the experiments required 600 GPU hours. Future work aims to balance effectiveness and computational demands.

Currently, we test classifiers across all languages due to limited ground truth, leading to bias. Future studies will address this issue.

We reduce inputs to 512 tokens to balance detail with resource limitations. Despite this, statistics indicate vulnerability fixes are often at file starts, and our model performs well even with truncated inputs, as confirmed in our experiment.

Time constraints limited our user study to a subset of scenarios with only three false positives. Future studies will involve more participants and tasks across different languages for a broader assessment of our approach.

External validity: The effectiveness of our models in different languages is uncertain due to ground truth's language mix. Although acquiring a diverse training set is difficult, our research is a starting point for future studies to enhance language-specific validity.

In our experiments, we tested only GPT-2-based models due to computational and financial constraints. Advanced models like GPT-3.5 [5] and GPT-4 [43] show greater promise in vulnerability-relevant file tracing, but their effective use requires well-crafted prompts, which is outside our current scope. These models also face issues with output instability and are still developing fine-tuning capabilities[5]. We plan to assess their effectiveness in future work.

6 RELATED WORK

Software Bill of Material: Limited research exists on SBOMs. Xia et al. [66] explored developers' views on SBOMs, noting issues like lack of design and awareness. Bi et al. [4] studied SBOM's role in open-source software (OSS), highlighting its necessity. Other studies focus on SBOM quality and usefulness [2, 3, 6, 41]. However, none specifically address SBOM construction for vulnerability detection, which our work aims to do, focusing on identifying vulnerability-relevant files in software packages.

Vulnerability-Relevant Package Tracing: To address the gap in vulnerability information in SBOMs, some studies use deep learning to link potential vulnerability-relevant packages to CVE records. Chen et al. [8] treated this as a multi-classification problem, while Haryono et al. [26] and Lyu et al. [36] improved this approach, considering cost and time. However, these methods mainly target package-level vulnerabilities, not finer-grained components.

Patching Commit Tracing: Various studies [17, 21, 28, 32, 40, 49, 63, 64, 67, 69, 71, 73–75] try to identify patching commits for CVEs to aid in understanding vulnerabilities. Methods range from manual searches to using NVD references and commit messages. Xu et al. [68] enhanced this by including external URLs, achieving higher recall. However, these approaches often don't pinpoint the exact vulnerability-relevant components. Our work advances this by tracing file-level components, offering more detailed information.

Vulnerable Code Tracing: Many studies aim to detect vulnerable functions or code lines, using either call-graph/static analysis or deep learning with large datasets [18, 27, 29–31, 46, 55, 77]. However, these methods don't typically trace CVE-relevant components, limiting access to key vulnerability analysis resources like MITRE ATT&CK [38], CWE [13], and CAPEC [11]. Our work, by focusing on tracing, complements these existing methods.

7 CONCLUSION

In our study, we developed a framework to trace files related to each disclosed vulnerability, using heuristic methods, large language models (LLMs), search engines, and a deep learning classifier to identify files for specific CVEs. We created a dataset of CVE-related files using CVE-patching commits and our heuristic method, and trained a classifier with a large dataset. Our framework showed high accuracy (99.7%) and the CodeBERT classifier performed well (AUC of 0.92, MAP of 0.85). A user study confirmed the approach's practicality and accuracy. Future work will focus on expanding the dataset to include various programming languages, improving its applicability in different scenarios.

[5]https://platform.openai.com/docs/guides/fine-tuning

REFERENCES

[1] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars!. In *Deep Learning for Code Workshop (DL4C)*.

[2] Iain Barclay, Alun Preece, Ian Taylor, Swapna Krishnakumar Radha, and Jarek Nabrzyski. 2022. Providing assurance and scrutability on shared data and machine learning models with verifiable credentials. *Concurrency and Computation: Practice and Experience* (2022), e6997.

[3] Iain Barclay, Alun Preece, Ian Taylor, and Dinesh Verma. 2019. Towards traceability in data ecosystems using a bill of materials model. *arXiv preprint arXiv:1904.04253* (2019).

[4] Tingting Bi, Boming Xia, Zhenchang Xing, Qinghua Lu, and Liming Zhu. 2023. On the Way to SBOMs: Investigating Design Issues and Solutions in Practice. *arXiv preprint arXiv:2304.13261* (2023).

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[6] Seth Carmody, Andrea Coravos, Ginny Fahs, Audra Hatch, Janine Medina, Beau Woods, and Joshua Corman. 2021. Building resilient medical technology supply chains with a software bill of materials. *npj Digital Medicine* 4, 1 (2021), 34.

[7] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[8] Yang Chen, Andrew E Santosa, Asankhaya Sharma, and David Lo. 2020. Automated identification of libraries from vulnerability data. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 90–99.

[9] Nancy Chinchor. 1992. MUC-4 Evaluation Metrics. In *Proceedings of the 4th Conference on Message Understanding (MUC4 '92)*. Association for Computational Linguistics, 22–29.

[10] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[11] Common Attack Pattern Enumeration and Classification. 2023. https://capec.mitre.org/. Accessed: 2023-07-31.

[12] Common Platform Enumeration. 2023. https://csrc.nist.gov/projects/security-content-automation-protocol/specifications/cpe. Accessed: 2023-07-31.

[13] Common Weakness Enumeration: CWE. 2023. https://cwe.mitre.org/. Accessed: 2023-07-31.

[14] CVE Request Template. 2023. http://cveproject.github.io/docs/content/key-details-phrasing.pdf. Accessed: 2023-07-31.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 4171–4186.

[16] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process* 25, 1 (2013), 53–95.

[17] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. 2019. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries. *Proceedings 2019 Network and Distributed System Security Symposium* (2019).

[18] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. FIRMSCOPE: Automatic uncovering of Privilege-Escalation vulnerabilities in Pre-Installed apps in android firmware. In *29th USENIX security symposium (USENIX Security 20)*. 2379–2396.

[19] English Wikipedia. 2023. https://en.wikipedia.org/wiki/Main_Page. Accessed: 2023-07-31.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, 1536–1547.

[21] 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.

[22] Sasi Kiran Gaddipati, Deebul Nair, and Paul G Plöger. 2020. Comparative evaluation of pretrained transfer learning models on automatic short answer grading. *arXiv preprint arXiv:2009.01303* (2020).

[23] GitHub Advisory Database. 2023. https://github.com/advisories. Accessed: 2023-07-31.

[24] Dan Goodin. 2021. The Internet's biggest players are all affected by critical Log4Shell 0-day. https://arstechnica.com/information-technology/2021/12/the-critical-log4shell-zero-day-affects-a-whos-who-of-big-cloud-services/. Accessed: 2023-07-31.

[25] James A Hanley and Barbara J McNeil. 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology* 143, 1 (1982), 29–36.

[26] Stefanus A Haryono, Hong Jin Kang, Abhishek Sharma, Asankhaya Sharma, Andrew Santosa, Ang Ming Yi, and David Lo. 2022. Automated identification of libraries from vulnerability data: Can we do better?. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 178–189.

[27] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 595–614.

[28] Triet Huynh Minh Le, David Hin, Roland Croft, and M Ali Babar. 2021. Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 717–729.

[29] Xin Li, Lu Wang, Yang Xin, Yixian Yang, and Yuling Chen. 2020. Automated vulnerability detection in source code using minimum intermediate representation learning. *Applied Sciences* 10, 5 (2020), 1692.

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2021. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2821–2837.

[31] 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.

[32] 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. In *25th Annual Network and Distributed System Security Symposium*.

[33] Jiahuei Lin, Bram Adams, and Ahmed E. Hassan. 2023. On the coordination of vulnerability fixes: An empirical study of practices from 13 CVE numbering authorities. *Empirical Software Engineering* 28, 151 (2023).

[34] Linux kernel. 2023. https://github.com/torvalds/linux. Accessed: 2023-07-31.

[35] 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).

[36] Yunbo Lyu, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Zhipeng Zhao, Xuan-Bach D. Le, Ming Li, and David Lo. 2023. Chronos: Time-aware zero-shot identification of libraries from vulnerability reports. In *Proceedings of the ACM/IEEE 45th International Conference on Software Engineering*. IEEE.

[37] Amir M Mir, Mehdi Keshani, and Sebastian Proksch. 2023. On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 201–211.

[38] MITRE ATT&CK. 2023. https://attack.mitre.org/. Accessed: 2023-07-31.

[39] National Vulnerability Database. 2023. https://nvd.nist.gov/. Accessed: 2023-07-31.

[40] 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.

[41] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, 2020, Proceedings 17*. Springer, 23–43.

[42] Open Source Insights. 2023. https://deps.dev/. Accessed: 2023-07-31.

[43] OpenAI. 2023. GPT-4 Technical Report. arXiv:cs.CL/2303.08774

[44] OWASP CycloneDX Software Bill of Materials (SBOM) Standard. 2023. https://cyclonedx.org/. Accessed: 2023-07-31.

[45] path-to-regexp - npm. 2023. https://www.npmjs.com/package/path-to-regexp. Accessed: 2023-07-31.

[46] Brian Pfretzschner and Lotfi ben Othmane. 2017. Identification of dependency-based attacks on node. js. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*. 1–6.

[47] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[48] S. C. R. Center. 2021. https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2021.pdf. Accessed: 2021-09-31.

[49] 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.

[50] G. Salton and M.J. McGill. 1983. *Introduction to Modern Information Retrieval*. McGraw-Hill.

[51] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[52] Security Bug Tracker. 2023. https://security-tracker.debian.org/tracker/. Accessed: 2023-07-31.

[53] Security Working Group. 2023. https://github.com/nodejs/security-wg. Accessed: 2023-07-31.

[54] server - npm. 2023. https://www.npmjs.com/package/server. Accessed: 2023-07-31.

[55] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. 2023. Silent spring: Prototype pollution leads to remote code execution in Node. js. In *USENIX Security Symposium 2023*.

[56] R.L. Singh and S.P. B. 1979. *Elements Of Practical Geography*. Kalyani Publishers.

[57] Snyk Vulnerability Database. 2023. https://snyk.io/vuln. Accessed: 2023-07-31.

[58] Jiamou Sun, Zhenchang Xing, Qinghua Lu, Xiwei Xu, Liming Zhu, Thong Hoang, and Dehai Zhao. 2023. Silent Vulnerable Dependency Alert Prediction with Vulnerability Key Aspect Explanation. In *Proceedings of the ACM/IEEE 45th International Conference on Software Engineering*. IEEE.

[59] Jiamou Sun, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2022. Heterogeneous Vulnerability Report Traceability Recovery by Vulnerability Aspect Matching. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 175–186.

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[61] Julian Von der Mosel, Alexander Trautsch, and Steffen Herbold. 2022. On the validity of pre-trained transformers for natural language processing in the software engineering domain. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1487–1507.

[62] Vulnerability Database - OSV. 2023. https://osv.dev/list. Accessed: 2023-07-31.

[63] Wenbo Wang, Tien N Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. 2023. DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2249–2261.

[64] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 35–45.

[65] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 38–45.

[66] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. 2023. An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead. In *Proceedings of the ACM/IEEE 45th International Conference on Software Engineering*. IEEE.

[67] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. 2020. {MVP}: Detecting Vulnerabilities using {Patch-Enhanced} Vulnerability Signatures. In *29th USENIX Security Symposium (USENIX Security 20)*. 1165–1182.

[68] Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. 2022. tracer: finding patches for open source software vulnerabilities. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, 860–871.

[69] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. 2020. Automatic Hot Patch Generation for Android Kernels. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2397–2414.

[70] Jeong Yang, Young Lee, and Arlen P McDonald. 2021. SolarWinds Software Supply Chain Security: Better Protection with Enforced Policies and Technologies. In *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. Springer, 43–58.

[71] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-Based Automatic Generation of Proof-of-Concept Exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. 2139–2154.

[72] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. 2018. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 559–563.

[73] Yaqin Zhou and Asankhaya Sharma. 2017. Automated Identification of Security Issues from Commit Messages and Bug Reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, 914–919.

[74] Yaqin Zhou and Asankhaya Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 914–919.

[75] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. 2021. Spi: Automated identification of security patches via commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–27.

[76] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*. 19–27.

[77] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2019), 2224–2236.