International Journal of Information Security (2024) 23:3311–3327
https://doi.org/10.1007/s10207-024-00888-y

REGULAR CONTRIBUTION

A comprehensive analysis on software vulnerability detection datasets: trends, challenges, and road ahead

Yuejun Guo[1] · Seifeddine Bettaieb[1] · Fran Casino[2,3]

Published online: 23 July 2024

Abstract
As society's dependence on information and communication systems (ICTs) grows, so does the necessity of guaranteeing the proper functioning and use of such systems. In this context, it is critical to enhance the security and robustness of the DevSecOps pipeline through timely vulnerability detection. Usually, AI-based models enable desirable features such as automation, performance, and efficacy. However, the quality of such models highly depends on the datasets used during the training stage. The latter encompasses a series of challenges yet to be solved, such as access to extensive labelled datasets with specific properties, such as well-represented and balanced samples. This article explores the current state of practice of software vulnerability datasets and provides a classification of the main challenges and issues. After an extensive analysis, it describes a set of guidelines and desirable features that datasets should guarantee. The latter is applied to create a new dataset, which fulfils these properties, along with a descriptive comparison with the state of the art. Finally, a discussion on how to foster good practices among researchers and practitioners sets the ground for further research and continued improvement within this critical domain.

Keywords Software vulnerability detection · Benchmarking · Datasets · DevSecOps

1 Introduction

Nowadays, cyberattacks related to security vulnerabilities are growing in terms of sophistication and number [1, 2]. In this context, vulnerability detection in source code is paramount to safeguard software applications against security threats. As evidenced by the Common Vulnerabilities and Exposures (CVE) Details database [3], 29065 vulnerabilities were reported in 2023, and more than 2000 were reported in early 2024, which shows that the continuous growth rate exhibited in previous years is far from ending.

✉ Fran Casino
franciscojose.casino@urv.cat

Yuejun Guo
yuejun.guo@list.lu

Seifeddine Bettaieb
seifeddine.bettaieb@list.lu

[1] Luxembourg Institute of Science and Technology (LIST), 5 Av. des Hauts-Fourneaux, Esch-Belval, Luxembourg

[2] Department of Computer Engineering and Mathematics, Universitat Rovira i Virgili, Tarragona, Spain

[3] Information Management Systems Institute, Athena Research Centre (ARC), Artemidos 6, Marousi, Greece

Traditional approaches to vulnerability detection, exemplified by rule-based methods [4] and signature-based techniques [5], typically use predefined rules or patterns indicative of known vulnerabilities. While effective in certain contexts, these approaches are frequently time-consuming to develop and may need to be adjusted to identify novel or previously unknown vulnerabilities. Such methods target specific known vulnerabilities, such as buffer overflows, SQL injection, or cross-site scripting. In the last years, machine learning [6], and in particular deep learning (DL) emerged as a promising alternative [7], leveraging its inherent capability to autonomously discern intricate patterns and features from vast amounts of source code [8–10]. The latter is leveraged by the advent of generative AI, with large language models (LLMs) indicating promising results [11–15].

Nevertheless, the efficacy of AI-based vulnerability detection encounters a significant limitation-the availability of high-quality benchmark datasets [6, 16, 17]. As AI models require vast amounts of data during the training stage, creating quality datasets directly impacts the efficacy of these model's parameter tuning. Despite numerous datasets being created and open-sourced in the literature (see Table 1 for more details), the quality of these datasets often falls short, potentially leading to inaccurate, biased, or incomplete

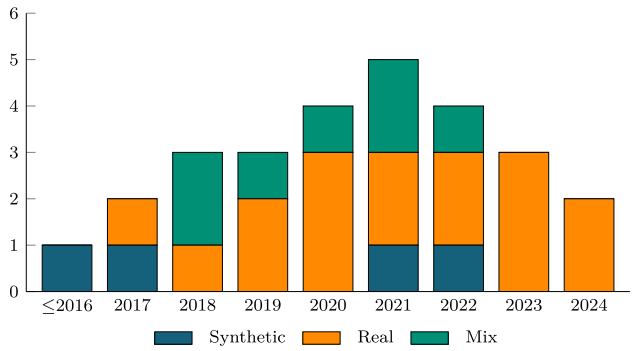

Fig. 1 Year-wise distribution of articles providing vulnerability datasets and their types. The term "Mix" denotes the use of both synthetic and real datasets

outcomes when used to train models for vulnerability detection. A frequent issue, for instance, is the low coverage of vulnerability types, with many datasets addressing only a handful of vulnerabilities (refer to Sect. 4.2).

After an exhaustive analysis of the state of the art, which includes the analysis of 92 vulnerability detection datasets extracted from 27 articles, our contribution highlights existing issues within such datasets and provides recommendations to overcome them. The latter expands the understanding of vulnerability detection datasets' intricacies and contributes to mitigating existing challenges by analysing them. Finally, we apply such analysis as a guideline to create a new benchmark dataset, and compare it with the state of the art in terms of features, showcasing its benefits.

The rest of this article is organised as follows. Section 2 provides the reader with the relevant background. Section 3 explains the search methodology followed to collect the articles used in Sect. 4 to provide a classification and description of the main findings related to existing datasets and their limitations. Section 5 describes our dataset creation guidelines and presents and example dataset crafted by us. Section 6 discusses the results of our analysis. Section 7 presents an overview of the related work. Finally, Sect. 8 concludes the article and identifies potential directions for future work.

2 Background

Vulnerability detection has become an increasingly important component of software security, enabling developers and security professionals to identify potential vulnerabilities more quickly and accurately than manual testing. Typically, AI-based vulnerability detection uses a model (e.g., a DL model), and describes the problem as a binary classification task where an input code is classified into two classes (e.g., either vulnerable or secure) [18, 19]. During the training procedure, labelled source code is fed into the model to tune its parameters to create a suitable mapping of inputs (source code) to outputs (vulnerable or secure predictions). In this setup, DL models automatically extract patterns of vulnerability, unlike traditional rule-based approaches [4]. As of today, various DL models have been developed and proven to perform effectively. Zhou et al. [8] proposed the graph neural network-based model Devign that embeds source code into graph data structures to learn patterns better. The large-scale pre-trained model CodeBERT [9] processes source code as plain text and supports multiple programming languages, such as Python, Java, and Javascript. Due to the dynamic creation of LLMs through fine-tuning procedures [11–14], we refer the reader to other sources such as Huggingface [20] for more on AI models capable of detecting vulnerable code and bad programming practices.

The effectiveness of AI-based vulnerability detection models highly depends on the data quality used for training [45]. By quality, we refer to the completeness of data and its verifiable provenance, consistency, orderliness (i.e., data should be organised and use standard notation and representation), and correctness [46]. Jimenez et al. [47] demonstrated that noisy historical data that is labelled secure but holds undiscovered vulnerabilities can cause the detection accuracy to decrease by over 20%. Garg et al. [48] confirmed that considering the noisy historical data can help improve a model's performance on vulnerability prediction. Note that if the data is incomplete, inconsistent, or biased, the models may produce unreliable results, leading to a false sense of security and wasted resources. Therefore, ensuring the quality of data is essential for the success of vulnerability detection.

3 Search methodology

Given the issues observed in the current state of the art in software vulnerability detection datasets, we wanted to further study them. Thus, we planned our review by using various features of the approach presented in [49]. First, we queried Scopus and Web of Science (WoS) with the following query TITLE-ABS-KEY (((vulnerability AND detection AND dataset AND software))) on June 2024 without time restriction. Next, we selected articles with the following criteria "Articles that are focused on software vulnerability detection and provide/create a dataset as part of their contribution", and applied the snowball effect to find further relevant literature by searching the references of key articles and reports for additional citations [50]. After thorough screening, we ended up with a total of 27 articles, providing a total of 92 unique datasets. Note that the information provided in such articles was used to classify the issues in existing datasets for AI-based vulnerability detection. The distribution of year publication can be seen in Fig. 1. More details about the codes and the sources can be seen in Table 1, and a specific analysis of each dataset is provided in Sect. 6.

Table 1 Overview of collected datasets. **real**: source code collected from real-world projects. **synthetic**: artificially generated code. **mix**: real+synthetic. **PL**: programming language

References	Year	Type	Granularity[1]	PL	Availability
[21]	2014	Synthetic	File	Java	Provided link not accessible anymore: https://sites.google.com/site/textminingandroid/
[18]	2017	Real	Function	C	https://github.com/DanielLin1986/function_representation_learning
[22]	2017	Synthetic	Function	C	https://github.com/mjc92/buffer_overrun_memory_networks
[23]	2018	Real	Function	C	https://github.com/DanielLin1986/TransferRepresentationLearning
[24]	2018	Mix	Program slice	C/C++	https://github.com/CGCL-codes/VulDeePecker
[25]	2018	Mix	Binary function	C	https://github.com/dascimal-org/MDSeqVAE
[8]	2019	Real	Function	C	https://sites.google.com/view/devign
[26]	2019	real	program slice	C	https://github.com/muVulDeePecker/muVulDeePecker
[27]	2019	Mix	Slice	C/C++	https://github.com/VulDeePecker/Comparative_Study
[28]	2020	Real	Function	C/C++	https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset
[29]	2020	Real	Function	C	https://github.com/Seahymn2019/Function-level-Vulnerability-Dataset
[30]	2020	Real	Project	Python	Not provided
[31]	2020	Mix	Function	C/C++	https://osf.io/d45bw/
[32]	2021	Mix	Program slice	C/C++	https://github.com/SySeVR/SySeVR
[33]	2021	Real	Trace	C	https://developer.ibm.com/exchanges/data/all/d2a/
[34]	2021	Mix	Function	C	https://github.com/DanielLin1986/RepresentationsLearningFromMulti_domain
[35]	2021	Synthetic	File	C/C++	Not provided
[9]	2021	Real	Function	C	https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection/dataset
[36]	2022	Real	Function	C/C++	https://github.com/VulDetProject/ReVeal
[37]	2022	Real	File, function	C/C++	Access to data expired: https://cybercodeintelligence.github.io/CyberCI/
[38]	2022	Synthetic	File	C/C++	Not provided
[39]	2022	Mix	Program	C/C++	https://github.com/MVDetection/MVD
[40]	2023	Real	Function	C/C++	https://github.com/wagner-group/diversevul
[41]	2023	Real	Function	Python	Available on request at: https://github.com/SunLab-GMU/PySecDB
[42]	2023	Real	Function	C/C++	https://zenodo.org/records/7123322
[43]	2024	Real	Function	C/C++	https://github.com/DLVulDet/PrimeVul
[44]	2024	Real	Function	C/C++	https://github.com/ttrangnguyen/CodeJIT/tree/main

Code granularity refers to the level of granularity at which code is labelled or segmented in the dataset

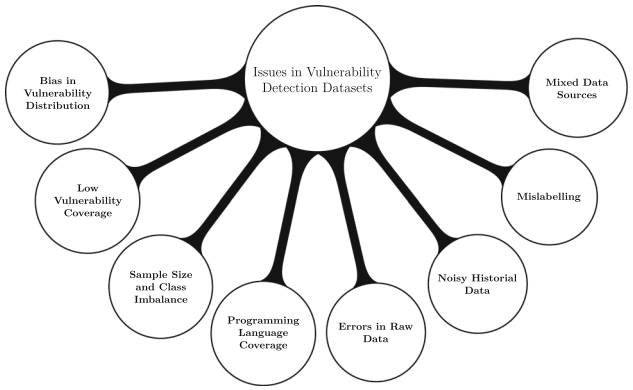

Fig. 2 Mindmap of the main vulnerability dataset issues found in the literature

4 Issues in existing datasets for vulnerability detection

This section classifies the main issues found in the literature, which are represented in Fig. 2. For instance, small sampling size, data imbalance, programming language coverage and low coverage of vulnerability types, and bias in vulnerability distribution, can markedly degrade model performance and generalisation and pose non-trivial challenges. Issues such as encompassing errors in raw data, mislabelling of source code, and the inclusion of noisy historical data, also have an impact on the model, and are not easily identifiable as such information may be missing from the source.

Some of the issues can be alleviated through pre-processing techniques. Such techniques include common operations to prepare data before training models, such as cleaning data to ensure uniqueness, removing comments and empty lines, writing data into a unified format (e.g., JSON and PKL files), verifying the labels of source code, and splitting data for training, validation, and testing.

4.1 Small sampling size and data imbalance

Training a model requires a minimum amount of data. However, quantifying such data is often impossible, as it depends on the model trained, its parameters, and the task. Usually, the richness of data representing the possible feature combinations for each class and the model's performance is used as a guideline to assess the quality of the training. Existing datasets often share issues regarding the number of samples, which translates into the above problems. In addition, these datasets exhibit an imbalance of data samples (i.e., the ratio between benign source code samples and vulnerable ones), which has been proven to be a recalling issue in the literature [53–55].

Table 3 describes the characteristics of 70 vulnerability detection datasets provided by 24 articles[1] (please refer to [56] for additional material). As it can be observed, the datasets LibPNG and LibTIFF in [18, 23] contain a low number of samples. More concretely, LibPNG only includes 621 source code samples in total. Concerning the sampling bias, except the Synthetic dataset by [22], Windows and Linux datasets by [25], and FFmpeg dataset by [8], all datasets contain a higher percentage of secure code samples than vulnerable ones. Particularly, the imbalance ratio is extreme for the Asterisk dataset by [23], obtaining a value of 324.82.

4.2 Low coverage of vulnerability types

Vulnerability detection models are expected to detect a broad range of vulnerability types, including common ones such as denial of service (DoS) and cross-site scripting (XSS), as well as less common ones such as HTTP response splitting and cross-site request forgery (CSRF) vulnerabilities [3]. However, existing datasets often have limited coverage of vulnerability types, which can lead to high rates of false negatives or an overall poor detection rate.

To showcase that situation, we summarise in Table 4 the vulnerability types of a total of 18 datasets aggregating the data reported in [18, 23, 29, 34]. In total, 28 vulnerability types are covered across these 18 datasets. These vulnerabilities belong to 10 vulnerability families[3], bypass something, XSS, DoS, directory traversal, code execution, gain privileges, overflow, gain information, SQL injection, and file inclusion. Other vulnerability families, such as HTTP response splitting and CSRF, are not covered by any dataset. In addition, note that each vulnerability family includes several vulnerability types, while in the list, most vulnerability types belong to the DoS (ID: 4-13) and the code execution (ID: 5-8, 15-23) vulnerability families.

4.3 Bias in vulnerability distribution

Vulnerability distribution refers to the numerical proportion of vulnerability types in a given dataset. Bias in the distribution could force a vulnerability prediction model to learn more from the majority types during the training procedure. Thus, the model will perform poorly on minority types. While this matter could be partially alleviated by creating subsets of a dataset with an equal number of samples per class, datasets often come with poorly represented vulnerability types, which could entail the creation of very small, balanced datasets. Therefore, such a situation could hinder the training procedures due to the presence of both imbalance and insufficient number of samples in some cases.

[1] Note that one article may provide more than one dataset.

Table 2 Labelling method of datasets provided by each reference

References	Data source	labelling method
[21]	F-Droid [51]	Use the automated static code analysis tool: HP FortifySCA
[18]	GitHub	Manual labelling based on NVD description
[22]	Generated based on Juliet Test Suite 1.1 [52]	Manual labelling using CWE
[23]	GitHub	Manual labelling based on NVD description
[24]	Synthetic: SARD; real: GitHub	Manual labelling using CWE; based on NVD description
[25]	Generated from [24]	Same as [24]
[8]	GitHub	Manual labelling using commit message based on NVD
[26]	SARD; open-source projects	Manual labelling using CWE; using commit message based on NVD
[27]	Synthetic: SARD; real: GitHub	Manual labelling using CWE; using commit message based on NVD
[28]	GitHub	Manual labelling using commit message based on CVE
[29]	GitHub	Manual labelling based on NVD description
[30]	GitHub	Manual labelling using commit message based on CVE and NVD
[31]	Synthetic: SATE IV Juliet Test Suite; real: ebian packages and GitHub	Static analysis, dynamic analysis, and commit-message/bug-report tagging
[32]	SARD and open-source projects	Same as [26]
[33]	Open-source projects	The proposed differential static analysis method
[34]	Synthetic: SARD; real: GitHub	Same as [24]
[9]	Data are combined from [8]	Same as [8]
[35]	Data filtered from SARD	Manual labelling using CWE from SARD
[36]	Bug repository Bugzilla and Debian security tracker	Manual labelling based on the instance tag, e.g., “bug”, “security”
[37]	GitHub	Manual labelling using commit message based on NVD
[38]	Generated based on SATE IV Juliet and SARD	Unknown
[39]	SARD; CVE	SARD statements notes; commit message
[40]	GitHub	Manual labelling using commit message
[41]	Real: GitHub; synthetic: generated using the real one	Manual labelling using commit message based on CVE
[42]	GitHub	Manually labelling using commit message

Table 2 continued

References	Data source	labelling method
[43]	Combination of existing datasets (GitHub)	Improve labels of existing datasets by checking code formatting and NVD description
[44]	Combination of existing datasets (GitHub)	Same as [8]

Table 3 Number of vulnerable and secure code samples of different datasets. The datasets provided by [21], [35], [38] are omitted due to restricted access. The imbalance ration is included in the last column for the sake of completeness. $Imbalance\ Ratio = \frac{\#Secure}{\#Vulnerable}$

References	Dataset	#Vulnerable	#Secure	Imbalance ratio
[18]	FFmpeg	191	5565	29.14
	LibPNG	44	577	13.11
	LibTIFF	94	731	7.78
[22]	Synthetic	7054	6946	0.98
[23]	Asterisk	56	18190	324.82
	FFmpeg	213	5552	26.07
	LibPNG	44	577	13.11
	LibTIFF	96	731	7.61
	Pidgin	29	8821	304.17
	VLC	42	6320	150.48
[24]	CGD	17725	43913	2.48
[25]	Windows	8978	8999	1.00
	Linux	7349	6955	0.95
[8]	FFmpeg	9679	4788	0.49
	QEMU	7479	10070	1.35
[26]	MVD	43119	137861	3.20
[27]	SARD	2669	0	–
	NVD	404	309	0.76
[28]	Big-Vul	10900	177736	16.31
[29]	LibPNG	45	577	12.82
	Pidgin	29	8626	297.45
	VLC	44	6115	138.98
[30]	FFmpeg	908	0	–
	ImageMagick	756	0	–
	OpenSSL	132	0	–
	PHP-SRC	426	0	–
	Linux	1716	0	–
[31]	Draper	82411	1191955	14.46
[32]	API function call	13603	50800	3.73
	Arithemetic expression	3475	18679	5.38
	Array usage	10926	31303	2.87
	Pointer usage	28391	263450	9.28
[33]	Httpd	217	12086	55.70
	Nginx	421	17945	42.62
	LibTIFF	553	12090	21.86
	OpenSSL	8022	342741	42.73
	Libav	4548	233275	51.29
	FFmpeg	4797	648104	135.11

Table 3 continued

References	Dataset	#Vulnerable	#Secure	Imbalance ratio
[34]	Asterisk	94	17755	188.88
	FFmpeg	249	5552	22.30
	LibTIFF	123	731	5.94
	LibPNG	45	577	12.82
	Pidgin	29	8626	297.45
	VLC	44	6115	138.98
[9]	Devign	12460	14858	1.19
[36]	ReVeal	2240	20494	9.15
[37]	Asterisk	94	17755	188.88
	FFmpeg	249	5552	22.30
	HTTPD	57	3850	6.75
	LibPNG	45	577	12.82
	LibTIFF	123	731	5.94
	OpenSSL	159	7068	44.45
	Pidgin	29	8626	297.45
	VLC	44	6115	138.98
	Xen	671	9023	13.45
[39]	Asterisk	94	0	–
	FFmpeg	272	0	–
	Libarchive	4	0	–
	Libav	12	0	–
	LibPNG	21	0	–
	LibTIFF	123	0	–
	Linux Kernel	662	0	–
	QEMU	36	0	–
	Wireshark	271	0	–
[40]	DiverseVul	18945	311547	16.44
[41]	PySecDB	1258	2791	2.22
[42]	Redis	82	1233	15.04
	Lua	32	750	23.44
[43]	PrimeVul	6968	228800	32.84
[44]	CodeJIT	8975	11299	1.26

Figure 3 shows the vulnerability distribution of six datasets provided by [23] (please refer to [56] for more results). The first observation is that each dataset covers a different set of vulnerability types, recalling the coverage issue discussed in Sect. 4.2. Second, regardless of the dataset, vulnerabilities are distributed unevenly. Remarkably, in FFmpeg, 9 out of 15 vulnerability types occupy less than 1% of source code from the entire dataset.

4.4 Mixed data sources

Data for vulnerability detection can come from multiple sources, which can introduce inconsistencies and biases. For instance, the source code may exhibit different coding styles from different software developers. In addition, each source has its specific characteristics and vulnerabilities, as shown in Fig. 3e, making it difficult to generalise across different sources. Using data from multiple sources can also result in data that is difficult to integrate and may require extensive pre-processing. An example of a dataset using mixed data sources is the Devign dataset provided by [9], which is a mixture of the FFmpeg and QEMU datasets by [8] (see Table 2).

4.5 Mislabelling on source code

Mislabelling can severely degrade the performance of prediction models. Generally, code files are extracted from open-source projects from GitHub, and labels are manually given based on commit messages and descriptions in

Table 4 List of vulnerability types included in datasets analysed in the referenced articles

No.	CWE-ID	Description	[18]	[23]	[29]	[34]
1	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	✓	✓	✓	✓
2	CWE-189	Numeric Errors	✓	✓	✓	✓
3	NVD-CWE-noinfo	Insufficient Information	✓	✓	✓	✓
4	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization	✓	✓	✗	✓
5	NVD-CWE-Other	Other	✓	✓	✓	✓
6	CWE-20	Improper Input Validation	✓	✓	✓	✓
7	CWE-295	Improper Certificate Validation	✓	✗	✗	✗
8	CWE-17	DEPRECATED	✓	✓	✗	✓
9	CWE-399	Resource Management Errors	✓	✓	✓	✓
10	CWE-191	Integer Underflow	✓	✓	✗	✓
11	CWE-125	Out-of-bounds Read	✓	✓	✓	✓
12	CWE-120	Buffer Copy without Checking Size of Input	✓	✓	✓	✓
13	CWE-190	Integer Overflow or Wraparound	✓	✓	✓	✓
14	CWE-787	Out-of-bounds Write	✓	✓	✓	✓
15	CWE-476	NULL Pointer Dereference	✓	✓	✓	✓
16	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	✓	✓	✓	✓
17	CWE-400	Uncontrolled Resource Consumption	✓	✓	✓	✓
18	CWE-834	Excessive Iteration	✓	✓	✗	✓
19	CWE-94	Improper Control of Generation of Code	✓	✓	✓	✓
20	CWE-79	Improper Neutralization of Input During Web Page Generation	✓	✓	✓	✓
21	CWE-401	Missing Release of Memory after Effective Lifetime	✓	✓	✓	✓
22	CWE-772	Missing Release of Resource after Effective Lifetime	✓	✓	✗	✓
23	CWE-254	7PK - Security Features	✓	✓	✗	✓
24	CWE-770	Allocation of Resources Without Limits or Throttling	✓	✓	✗	✓
25	CWE-369	Divide By Zero	✓	✓	✓	✓
26	CWE-416	Use After Free	✓	✗	✓	✓
27	CWE-134	Use of Externally-Controlled Format String	✗	✓	✓	✓
28	CWE-287	Improper Authentication	✗	✓	✗	✓
29	CWE-264	Permissions, Privileges, and Access Controls	✗	✓	✓	✓
30	CWE-284	Improper Access Control	✗	✓	✗	✓
31	CWE-835	Loop with Unreachable Exit Condition	✗	✓	✗	✓
32	CWE-617	Reachable Assertion	✗	✓	✗	✓
33	CWE-22	Improper Limitation of a Pathname to a Restricted Directory	✗	✓	✓	✓
34	CWE-824	Access of Uninitialized Pointer	✗	✗	✓	✓
35	CWE-310	Cryptographic Issues	✗	✗	✗	✓
36	CWE-78	Improper Neutralization of Special Elements used in an OS Command	✗	✗	✗	✓
37	CWE-459	Incomplete Cleanup	✗	✗	✗	✓
38	CWE-754	Improper Check for Unusual or Exceptional Conditions	✗	✗	✗	✓
39	CWE-129	Improper Validation of Array Index	✗	✗	✗	✓
In total			**26**	**31**	**22**	**38**

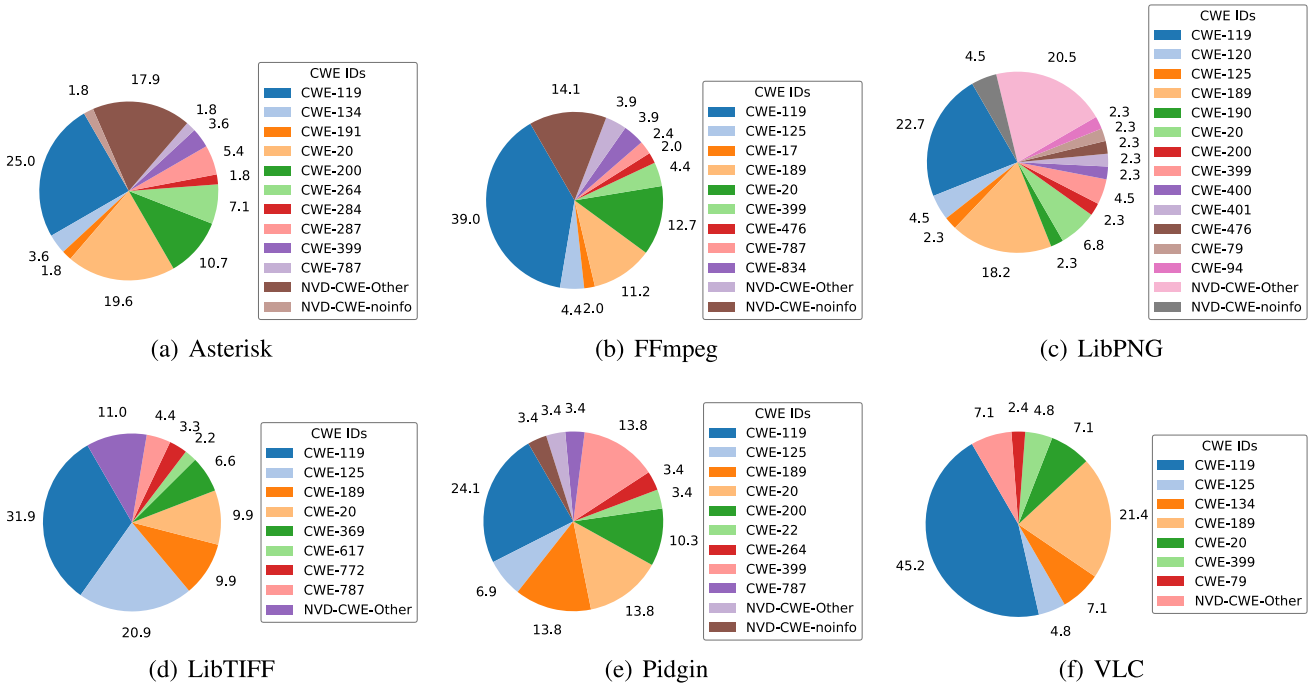

Fig. 3 Vulnerability distribution in each dataset provided by [23]. The numeral text indicates the percentage of source code. For the sake of clarity, we avoided showing CWEs representing less than 1% of the dataset

```
711      -   png_debug1(1, "in %s storage function", (png_ptr->chunk_name[0] == '\0' ?
         711 +   png_debug1(1, "in %s storage function", ((png_ptr == NULL ||
         712 +       png_ptr->chunk_name[0] == '\0') ?
712      713         "text" : (png_const_charp)png_ptr->chunk_name));
```

Fig. 4 An example of mislabelling

NVD[5]. Both commit messages and NVD entries are manually curated and analysed, which is error-prone even with experienced developers [57]. Another common way is to use static code analysis tools (Table 1), which is less accurate than the manual manner [58].

Figure 4 shows an example of mislabelling in the LibPNG dataset provided by [23]. The code file (including green lines in the example) named "cve-2016-10087.c" indicates that the source code is vulnerable, allowing context-dependent attackers to cause a NULL Pointer Dereference vectors. However, according to the commit message on GitHub,[2] the green lines are the patch of the corresponding vulnerable code (in red), thus, the label should be secure instead of vulnerable.

4.6 Noisy historical data

Noisy historical data [47, 48] refers to the phenomenon that code labelled as secure might be identified as vulnerable in the future, given that most vulnerabilities are discovered much later than when they are introduced (e.g. zero-day vulnerabilities). For example, the `decode_main_header()` function in FFmpeg is recently reported to have the null pointer dereference flaw.[3] However, when collecting data before such a report, this function will appear as secure.

4.7 Errors in raw data

Errors in raw data can significantly affect the accuracy of vulnerability detection algorithms. We have found errors in existing datasets, including empty source code files, extra lines, and inconsistent file formats. [23] includes 435, 195, and 205 empty source code files in Asterisk, Pidgin, and VLC, respectively. Without their identification, these empty files would be processed normally to train a detection model. However, since they do not provide any pattern, the detection model to be trained can be misled to learn real patterns for secure and vulnerable source code.

All the 18 datasets provided by [18], [23], [29], and [34] have the issue of containing an extra line at the beginning of a separate source code file. This extra line varies across different datasets, such as [34]: such as "`} EightBpsContext;`", "`}`", "`*/`", and "`} AascContext;`". Note that this extra line cannot be removed during the pre-processing procedure without checking the source code files manually. For example, the data pre-processing procedure generally filters

[2] https://github.com/glennrp/libpng/commit/a4d439b97507b54d7f08543e03eb8f006ea73bc5?diff=unified

[3] https://nvd.nist.gov/vuln/detail/CVE-2022-3341

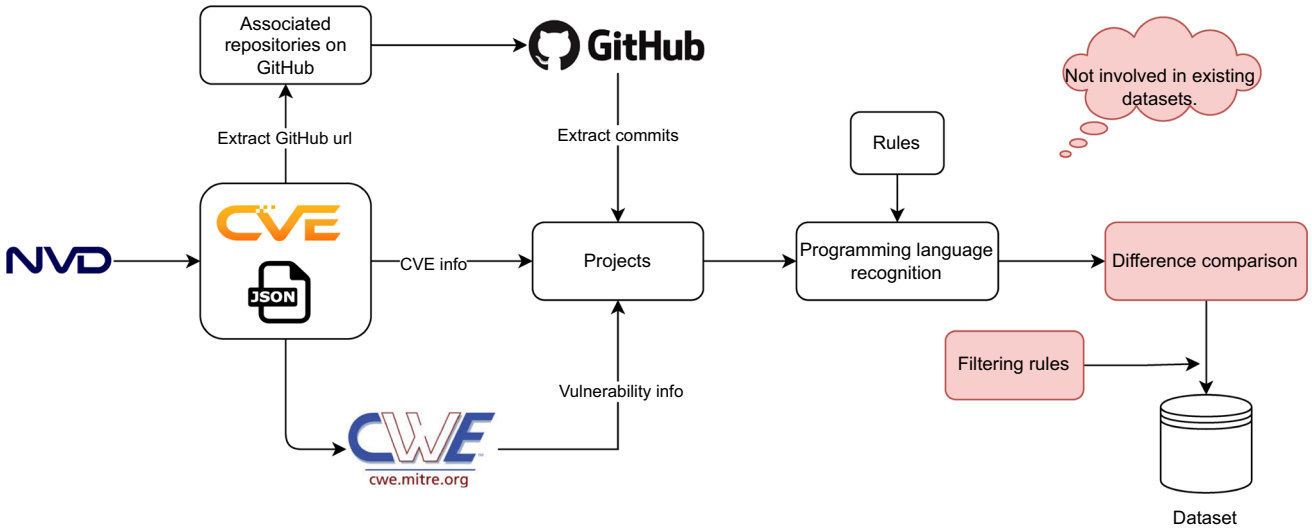

Fig. 5 Overview of the collection procedure of our dataset

comments by locating paired comment marks, such as "\\" and "\ ∗ ∗/" for C programming language. The work presented in [18] uses ".txt" for vulnerable code files and ".c" for secure code files. Depending on the framework, reading files can encounter errors.

4.8 Covered programming Languages

Existing datasets usually involve specific programming languages, such as C/C++, Java, and Python. Table 1 demonstrates an abundance of data in C and C++ and, a staggering absence of Java and Python despite them being two of the most prominent programming languages. This limitation leads to a gap in evaluating a detection model's generalisation according to different programming languages.

5 Our dataset creation methodology

For the sake of completeness, we created a dataset providing all the features described in Table 7 and open-sourced it on Zenodo [59]. We collected vulnerable source code from projects on GitHub that have registered CVEs into NVD[5] from 2002 to 2023. Figure 5 illustrates the workflow of our data collection process. First, we collect all the CVE records of each year from the NVD Data JSON Feeds [60]. Each CVE record consists of the detailed information, such as CVE, impact, and references. Second, we identify the associated GitHub repository (if any) along with the commit hash based on the URL filed in the reference data of each CVE record. Next, based on the commit message, we extract the code files before and after this commit. From CWE.mitre, we obtain the vulnerability type and detailed information for each CVE record. Once code files in GitHub projects are collected, we identify the programming language using file extensions according to predefined rules. These rules are dedicated to assign specific file extensions to all programming languages known to GitHub (in total 484 programming languages), ensuring accurate identification throughout our analysis process.

Compared to most existing datasets [8, 18, 23–25, 27, 28] that simply assign code before commit as vulnerable and after as non-vulnerable, we compare code difference and apply several filtering rules to avoid mislabelling and duplication. Specifically, we compare the code difference before and after commit and apply several filtering rules to ensure a high-quality dataset. These rules include:

1. Eliminating duplicated code. Removing code that does not have substantive changes, usually caused by modifying code formatting.
2. Removing spurious entries. Filtering out instances where changes are limited to modifying variable or function names, adding or deleting comments, rather than making significant alterations to the code logic.

In addition, for each vulnerable code, we collect its available patch(es) to support repairing guidance. These patches are considered as secure code. Note that according to [48], it is recommended to utilise the most recent patch concerning the noise. Nevertheless, we provide access to all available patches, ensuring users have the flexibility to choose based on their specific requirements or preferences.

Our dataset covers eight programming languages, including JavaScript, C, PHP, Java, C++, Python, Go, and Ruby. Table 5 shows the data size for each programming language and Table 6 presents the coverage of 2023 CWE top 25 most

Table 5 Summary of our dataset. #with patch indicates at least one patch exists for a certain vulnerable code

No.	PL	#Vulnerable	#Secure	#with patch	#CWE	#CVE
1	JavaScript	79551	181534	70485	85	639
2	C	6766	14872	6185	117	2408
3	PHP	6696	26087	5888	91	997
4	Java	2923	3661	2643	82	367
5	C++	2431	4455	2321	86	634
6	Python	2404	2873	2139	99	469
7	Go	1472	17683	1327	84	259
8	Ruby	1410	1175	1055	59	224

dangerous software weaknesses [61]. More details about the features, committer, commit date, and projects can be found in the dataset description files [59]. Note that, this dataset will be updated to have newer versions according to new data collected from GitHub or other sources, formatted and revised periodically.

6 Discussion

According to the previous analysis, we have extracted a set of desired features to be present in vulnerability detection datasets. Table 7 summarises them, along with the corresponding mapping of the datasets analysed in this article. First, we consider provenance as a critical aspect of verifiability and pre-processing. In this regard, we can ensure that the dataset contains correct versions of the code (i.e., easing the analysis and correction of secondary issues), if there are novel code excepts that can be used to update the dataset, and that multiple sources can be cleaned and tagged accordingly, which translates into a more robust database. In terms of adaptability, including multiple programming languages increases the dataset's use cases and models' capability to identify vulnerable code in multiple contexts. The latter, while not being critical, is a desirable feature to ensure adoption in DevSecOps pipelines, where multiple users with different necessities coexist. Some qualitative aspects that could affect the accuracy of models are related to the size and richness of the dataset. A well-represented number of classes in terms of secure and vulnerable code and their balance are also crucial to guarantee sound training procedures. In parallel, the above should also apply to the set of vulnerabilities, which should include as many as possible types with enough samples to be fed into the model. Finally, regarding usability, using correct and standard formatting increases the potential use across platforms and models and the use of automated updating mechanisms. The latter could be enhanced by the use of guidelines on how to correct or repair such vulnerabilities and patches for the corresponding code samples.

In terms of minimising the impact of issues in current datasets, certain problems (Sect. 4.1 - 4.3) are difficult to address by data pre-processing. However, there are techniques to mitigate their impact on the model performance. For the sampling size issue, data augmentation [62] can be applied to increase the number of data during training. To force the model to learn more from vulnerable code in imbalanced datasets, weighted loss functions, such as focal loss [55] and mean squared error loss [53], can be applied instead of the default cross-entropy loss. Concerning the low vulnerability coverage issue, one can consider merging several datasets that cover different vulnerabilities or just focus on detecting source code with included vulnerabilities. Code refactoring [62] and adversarial code attacks [63] can help to generate more similar code samples without changing the semantics to reduce the bias in vulnerability distribution. Finally, despite the real-world prevalence of imbalanced datasets, such imbalance can hinder the robustness of the performance of learning systems [64], and thus, several balancing strategies can be used [65]. For instance, sub-sampling of the dataset can also be used to provide balanced instances, yet only if the dataset has a sufficient number of samples for all classes and types to guarantee quality training.

For other issues (Sect. 4.4 - 4.8), the quality of existing datasets can be improved by designing an advanced pre-processing method to remove errors, check the correctness of labels, and reduce noise from the raw data. Both static analysis tools and expert manual manner should be considered when assigning labels to collected data to avoid mislabelling. In addition, one should include as much information as possible in the dataset, such as vulnerability type, source project, and commit ID, instead of only including labels to allow for tracking and re-checking. Note that, the errors mentioned in Sect. 4.7 do not exist in all studied datasets and may not cover all cases. A thorough check should be made to develop the operations for a comprehensive pre-processing. To reduce noise, one should look into the latest commit messages related to each code file and modify the labels when

Table 6 Coverage of 2023 CWE top 25 most dangerous software weaknesses [61]

No.	CWE-ID	Description	JavaScript	C	PHP	Java	C++	Python	Go	Ruby
1	CWE-787	Out-of-bounds Write	✗	✓	✓	✓	✓	✓	✓	✗
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	✓	✓	✓	✓	✓	✓	✓	✓
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	✓	✓	✓	✓	✓	✓	✓	✓
4	CWE-416	Use After Free	✗	✓	✗	✗	✓	✗	✓	✗
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	✓	✓	✓	✓	✓	✓	✓	✓
6	CWE-20	Improper Input Validation	✓	✓	✓	✓	✓	✓	✓	✓
7	CWE-125	Out-of-bounds Read	✓	✓	✗	✓	✓	✓	✓	✗
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	✓	✓	✓	✓	✓	✓	✓	✓
9	CWE-352	Cross-Site Request Forgery (CSRF)	✓	✓	✓	✓	✓	✓	✗	✓
10	CWE-434	Unrestricted Upload of File with Dangerous Type	✓	✓	✓	✓	✓	✓	✗	✓
11	CWE-862	Missing Authorization	✓	✓	✓	✓	✗	✓	✓	✓
12	CWE-476	NULL Pointer Dereference	✗	✓	✗	✓	✓	✓	✓	✗
13	CWE-287	Improper Authentication	✓	✓	✓	✓	✓	✓	✓	✓
14	CWE-190	Integer Overflow or Wraparound	✓	✓	✓	✓	✓	✓	✓	✗
15	CWE-502	Deserialization of Untrusted Data	✓	✗	✓	✓	✓	✓	✗	✓
16	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	✓	✓	✓	✓	✓	✓	✓	✓
17	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	✗	✓	✗	✓	✓	✓	✓	✓
18	CWE-798	Use of Hard-coded Credentials	✗	✗	✓	✓	✗	✗	✓	✗
19	CWE-918	Server-Side Request Forgery (SSRF)	✓	✗	✓	✓	✓	✓	✓	✓
20	CWE-306	Missing Authentication for Critical Function	✗	✗	✗	✓	✗	✓	✓	✗
21	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	✗	✓	✓	✓	✓	✓	✓	✓
22	CWE-269	Improper Privilege Management	✓	✓	✓	✓	✓	✓	✓	✗
23	CWE-94	Improper Control of Generation of Code ('Code Injection')	✓	✓	✓	✓	✓	✓	✓	✓
24	CWE-863	Incorrect Authorization	✓	✓	✓	✓	✗	✓	✓	✓
25	CWE-276	Incorrect Default Permissions	✗	✓	✗	✗	✓	✗	✓	✓
In total			**17**	**21**	**19**	**23**	**21**	**22**	**21**	**17**

Table 7 Mapping of the desired features across the methodology used in the articles providing vulnerability datasets. The datasets provided by [21], [35], and [38] are omitted due to restricted access

References	Provenance	MPL	Balanced Samples	Well-represented Samples	Well-represented Vulnerabilities	Standard Formatting	Repairing Guidelines
[18]	✗	✗	✗	✗	✗	✗	✗
[22]	✗	✗	✗	✗	✗	✗	✗
[23]	✗	✗	✗	✗	✗	✗	✗
[24]	✗	✗	✗	✓	✗	✗	✓
[25]	✗	✗	✗	✓	✗	✗	✓
[8]	✗	✗	✗	✗	✗	✗	✗
[26]	✗	✗	✗	✗	✗	✗	✗
[26]	✗	✗	✗	✗	✓	✗	✓
[28]	✓	✗	✗	✓	✓	✓	✓
[29]	✗	✗	✗	✗	✗	✗	✗
[30]	-	✗	✗	-	-	-	✗
[31]	✗	✗	✓	✗	✗	✓	✗
[32]	✗	✗	✗	✗	✗	✗	✗
[33]	✓	✗	✗	✓	✓	✓	✗
[34]	✗	✗	✗	✗	✗	✗	✗
[9]	✗	✗	✗	✗	✗	✗	✗
[36]	✗	✗	✗	✗	✗	✓	✗
[37]	✗	✗	✗	✗	✗	✗	✗
[39]	✗	✗	✗	✗	✗	✗	✓
[40]	✓	✗	✗	✓	✓	✓	✗
[41]	✓	✗	✓	✗	✗	✗	✗
[42]	✗	✗	✗	✗	✗	✗	✗
[43]	✓	✗	✗	✗	✓	✗	✓
[44]	✗	✗	✗	✗	✗	✗	✗
Our dataset	✓	✓	✓	✓	✓	✓	✓

Multiple programming languages

necessary. However, the latter can only be done if the provenance of the data is properly managed and verified.

6.1 Limitations

As seen in Table 7, we crafted a dataset that fulfils all the desired features. Compared to most of the evaluated vulnerability datasets, a particular aspect is the inclusion of the desired feature of providing potential solutions to help developers, enhancing the DevSecOps pipeline. The latter is currently being exploited by AI-based models such as GitHub Copilot [66] and other LLMs [20]. Nevertheless, it is worth to mention some limitations of our approach. First, in terms of data collection, the sources used to retrieve data are GitHub repositories which have registered CVEs into NVD, potentially introducing bias and limiting the representation of diverse software vulnerabilities. Next, regarding code granularity, the dataset is constrained to the function-level code given its more frequent utilisation in the existing literature. Finally, certain programming languages, notably Go and Ruby, have smaller data sizes within this dataset than others due to the scarcity of sufficient repositories, potentially leading to imbalances in vulnerability distribution. Consequently, the model's efficacy in detecting vulnerabilities may be compromised, particularly when applied to source code written in these programming languages.

Regarding current regulations and models, Stanford's Center for Research on Foundation Models (CRFM) recently evaluated the major AI companies on their transparency [67]. The findings revealed a significant gap in the AI industry's transparency. Our dataset contains the provenance references of the code samples collected, providing data source transparency as required by the European AI ACT [68]. The latter, paired with the use of versioning through hashes, can provide additional protection against the use of adversarial data and similar mechanisms, as the contents of the dataset could be verified via tamper-proof mechanisms such as bloc-kchain. Similarly, we could enforce verified code when providing repairing/patching suggestions to avoid using malicious code. The latter is a research line that we will pursue in the future.

7 Related work

The quality issues in existing datasets have drawn the attention of researchers [69]. Most datasets with vulnerable code from real-world projects are extracted from publicly accessible repositories on GitHub. These datasets include vulnerabilities reported in databases like CVE,[4] the National Vulnerability Database (NVD),[5] and Exploit-DB.[6] It is important to note that these databases do not contain source code themselves; they provide vulnerability information and links to related projects (i.e. often referring to GitHub), from which the source code is retrieved. Generally, simulated datasets are extracted from the Software Assurance Reference Database (SARD)[7] where synthetic code is provided for various vulnerabilities. However, a recent comparison study [70] revealed that some databases, including NVD, may introduce errors because they do not perform vulnerability testing on reports. As a result, corresponding datasets may contain mislabelled data. For example, a recent study by Croft et al. [71] found that 20-71% of vulnerabilities are inaccurate in four existing datasets, and 17-99% data is duplicated.

Hanif et al. [72] pointed out three open issues of datasets for vulnerability detection datasets, including the lack of labelled datasets, inconsistencies of datasets for evaluation, and the impractical use of synthetic datasets since synthetic data cannot reflect the true structure of real-world vulnerabilities. Nie et al. [73] specifically studied the cause of label errors in existing datasets and investigated the impact on model performance. Croft et al. [71] studied five quality issues in existing datasets [8, 28, 33], including accuracy, uniqueness, consistency, completeness, and currentness. Accuracy refers to the label correctness and the causes of mislabelling are categorised into: irrelevant code changes, cleanup change, and inaccurate vulnerability fix identification. Uniqueness refers to the duplication of data samples. Consistency refers to duplicated data that have different labels, which is a type of mislabelling. Completeness refers to the reference information of data samples, and currentness focuses on if a dataset is up to date. All these five issues have been covered by our study. Ding et al. [43] revealed three issues in existing datasets [9, 28, 40], including low label accuracy, and high duplication rates.

In our article, we leverage a search methodology to provide a thorough state of the art analysis. As we collect all the previous efforts in terms of issue/challenge identification, we are able to provide a more fine-grained analysis of the existing datasets, as identified in the literature. The latter allowed us to identify and describe the current issues in a comprehensive manner, and to generate a new dataset that fulfils the identified desirable features while providing a quantitative comparison with the state of the art.

[4] https://www.cve.org/

[5] https://nvd.nist.gov/

[6] https://www.exploit-db.com/

[7] https://samate.nist.gov/SARD/

8 Conclusion

The performance of AI-based vulnerability detection models highly relies on the data used for training. Poor data quality can lead to unreliable results, false positives, and false negatives. In this article, we provide an in-depth discussion of existing vulnerability datasets and define eight quality issues. Furthermore, we provide actionable guidance to assist researchers in addressing these issues when using existing datasets and open-source a real-world dataset with all desired features extracted from our classification.

In future work, we will foster the pre-processing and updating capabilities of the datasets to avoid the issues discussed in our classification. We will also study the use of tamper-proof mechanisms and their impact on the trustworthiness and usability of the vulnerability detection datasets. The latter, paired with the adoption of novel regulations and guidelines such as the European AI Act, should guarantee the robustness and verifiability of datasets and training procedures, enhancing vulnerability detection practices.

Acknowledgements This work was supported by the European Commission under the Horizon Europe Programme, as part of the projects LAZARUS (Grant Agreement no. 101070303) and HEROES (Grant Agreement no. 101021801). This work was also supported by the European Union's Internal Security Fund as part of the CTC project (Grant Agreement no. 101036276) and ALUNA (Grant Agreement no. 101084929). Fran Casino was supported by the Government of Catalonia with the Beatriu de Pinós programme (Grant No. 2020 BP 00035), and by AGAUR with the project ASCLEPIUS (2021SGR-00111). The content of this article does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the authors.

Author Contributions main conceptualization - Y.G and S.B. methodology - Y.G and F.C. Implementation and experiments - Y.G figure preparation - Y.G. and F.C. manuscript writting and editing - All manuscript review - All

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

data availability The dataset generated during this study can be found in Zenodo [59]. Additional figures and materials can be found in [56].

Declarations

Conflict of interest The authors declare that they have no known competing financial interests or personal relationships that could appear to influence the work described in this paper.

Ethical approval The authors declare full compliance with ethical standards. This article does not contain any studies involving humans or animals performed by any of the authors.

References

1. Aslan, Ö., Aktuğ, S.S., Ozkan-Okay, M., Yilmaz, A.A., Akin, E.: A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions. Electronics **12**(6), 1333 (2023). https://doi.org/10.3390/electronics12061333
2. Casino, F., Dasaklis, T.K., Spathoulas, G.P., Anagnostopoulos, M., Ghosal, A., Borocz, I., Solanas, A., Conti, M., Patsakis, C.: Research trends, challenges, and emerging topics in digital forensics: A review of reviews. IEEE Access **10**, 25464–25493 (2022)
3. SecurityScorecard. CVE vulnerabilities by year. https://www.cvedetails.com/browse-by-date.php . Accessed on January 30th, 2024 (2024)
4. Lee, M., Cho, S., Jang, C., Park, H., Choi, E.: In *International Conference on Hybrid Information Technology*, vol. 2, pp. 505–512. (2006) https://doi.org/10.1109/ICHIT.2006.253653
5. Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A., Piras, L.: Android source code vulnerability detection: a systematic literature review. ACM Comput. Surv. **55**(9), 1–37 (2023). https://doi.org/10.1145/3556974
6. Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Vats, I., Moazen, H., Sarro, F.: A survey on machine learning techniques applied to source code. J. Syst. Softw. **209**, 111934 (2024)
7. Croft, R., Newlands, D., Chen, Z., Babar, M.A.: In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Association for Computing Machinery, New York, NY, USA,), ESEM '21. (2021) https://doi.org/10.1145/3475716.3475781
8. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: in *Proceedings of the 33rd International Conference on Neural Information Processing Systems* (Curran Associates Inc., Red Hook, NY, USA,), p. 10197–10207 (2019)
9. Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C.B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S.K., Fu, S., Liu, S.: CodeXGLUE: A machine learning benchmark dataset for code understanding and generation, CoRR arXiv:2102.04664 (2021)
10. Vouvoutsis, V., Casino, F., Patsakis, C.: On the effectiveness of binary emulation in malware classification. J. Inf. Secur. Appl. **68**, 103258 (2022)
11. Khare, A., Dutta, S., Li, Z., Solko-Breslin, A., Alur, R., Naik , M.: Understanding the effectiveness of large language models in detecting security vulnerabilities. https://arxiv.org/pdf/2311.16169.pdf (2023). Accessed on January 30th, (2024)
12. Fu, M., Tantithamthavorn, C., Nguyen, V., x Le, V.: Chatgpt for vulnerability detection, classification, and repair: how far are we? https://arxiv.org/pdf/2310.09810.pdf (2023). Accessed on January 30th, 2024
13. Purba, M.D., Ghosh, A., Radford, B.J., Chu, B.: In *IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)* IEEE Computer Society, Los Alamitos, CA, USA, (2023), pp. 112–119. https://doi.org/10.1109/ISSREW60843.2023.00058

14. Gao, Z., Wang, H., Zhou, Y., Zhu, W., Zhang, C.: How far have we gone in vulnerability detection using large language models. https://arxiv.org/pdf/2311.12420.pdf (2023)
15. Patsakis, C., Casino, F., Lykousas, N.: arXiv preprint arXiv:2404.19715 (2024)
16. Hanif, H., Nasir, M.H.N.M., Ab Razak, M.F., Firdaus, A., Anuar, N.B.: The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. J. Netw. Comput. Appl. **179**, 103009 (2021)
17. Guo, Y., et al.: In *European Symposium on Research in Computer Security* Springer, (2024). To appear
18. Lin, G., Zhang, J., Luo, W., Pan, L., Xiang, Y.: In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* Association for Computing Machinery, New York, NY, USA, (2017), CCS '17, p. 2539–2541. https://doi.org/10.1145/3133956.3138840
19. Marjanov, T., Pashchenko, I., Massacci, F.: Machine learning for source code vulnerability detection: What works and what isn't there yet. IEEE Secur. Priv. **20**(05), 60 (2022). https://doi.org/10.1109/MSEC.2022.3176058
20. AI community. Hugging face. https://huggingface.co/ (2024). Accessed on January 30th, (2024)
21. Scandariato, R., Walden, J., Hovsepyan, A., Joosen, W.: Predicting vulnerable software components via text mining. IEEE Trans. Softw. Eng. **40**(10), 993 (2014). https://doi.org/10.1109/TSE.2014.2340398
22. Choi, M.J., Jeong, S., Oh, H., Choo, J.: In *Proceedings of the 26th International Joint Conference on Artificial Intelligence* AAAI Press, (2017), IJCAI'17, p. 1546–1553
23. Lin, G., Zhang, J., Luo, W., Pan, L., Xiang, Y., De Vel, O., Montague, P.: Cross-project transfer representation learning for vulnerable function discovery. IEEE Trans. Ind. Inf. **14**(7), 3289 (2018). https://doi.org/10.1109/TII.2018.2821768
24. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: In *25th Annual Network and Distributed System Security Symposium (NDSS)* (The Internet Society, 2018). https://doi.org/10.14722/ndss.2018.23158. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-2_Li_paper.pdf
25. Le, T., Nguyen, T., Le, T., Phung, D., Montague, P., Vel, O.D., Qu, L.: In *International Conference on Learning Representations* (2019). https://openreview.net/forum?id=ByloIiCqYQ
26. Zou, D., Wang, S., Xu, S., Li, Z., Jin, H.: IEEE Transactions on Dependable and Secure Computing **PP**, 1 (2019). https://doi.org/10.1109/TDSC.2019.2942930
27. Li, Z., Zou, D., Tang, J., Zhang, Z., Sun, M., Jin, H.: Multi-Modal Medical Image Fusion WithAdaptive WeightedCombinationofNSSTBandsUsing Chaotic Grey Wolf Optimization. IEEE Access **7**, 103184 (2019). https://doi.org/10.1109/ACCESS.2019.2930578
28. Fan, J., Li, Y., Wang, S., Nguyen, T.N.: In *Proceedings of the 17th International Conference on Mining Software Repositories* Association for Computing Machinery, New York, NY, USA, (2020), MSR '20, p. 508–512. https://doi.org/10.1145/3379597.3387501
29. Lin, G., Xiao, W., Zhang, J.: Deep learning-based vulnerable function detection: A benchmark. Y. Xiang. In: Zhou, J., Luo, X., Shen, Q., Xu, Z. (eds.) Information and Communications Security, pp. 219–232. Springer International Publishing, Cham (2020)
30. Liu, B., Meng, G., Zou, W., Gong, Q., Li, F., Lin, M., Sun, D., Huo, W., Zhang, C.: In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* Association for Computing Machinery, New York, NY, USA, (2020), ICSE '20, p. 1547–1559. https://doi.org/10.1145/3377811.3380923
31. Li, X., Wang, L., Xin, Y., Yang, Y., Chen, Y.: Automated vulnerability detection in source code using minimum intermediate representation learning. Appl. Sci. **10**(5), 1692 (2020)
32. Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z.: Can large language models better predict software vulnerability? IEEE Trans. Depend. Secure Comput. **19**(04), 2244 (2022). https://doi.org/10.1109/TDSC.2021.3051525
33. Zheng, Y., Pujar, S., Lewis, B., Buratti, L., Epstein, E., Yang, B., Laredo, J., Morari, A., Su, Z.: In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice* IEEE Press, (2021), ICSE-SEIP '21, p. 111–120. https://doi.org/10.1109/ICSE-SEIP52600.2021.00020
34. Lin, G., Zhang, J., Luo, W., Pan, L., De Vel, O., Montague, P., Xiang, Y.: Software vulnerability discovery via learning multi-domain knowledge bases. IEEE Trans. Depend. Secure Comput. **18**(5), 2469 (2021). https://doi.org/10.1109/TDSC.2019.2954088
35. Ziems, N., Wu, S.: In *IEEE INFOCOM WKSHPS: The Ninth International Workshop on Security and Privacy in Big Data (BigSecurity 2021)* IEEE, (2021), pp. 1–6. https://doi.org/10.1109/INFOCOMWKSHPS51825.2021.9484500
36. Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: Are we there yet? IEEE Trans. Softw. Eng. **48**(09), 3280 (2022). https://doi.org/10.1109/TSE.2021.3087402
37. Yuan, X., Lin, G., Tai, Y., Zhang, J.: Deep neural embedding for software vulnerability discovery: Comparison and optimization. Secur. Commun. Netw. **2022**(1), 5203217 (2022)
38. Zhou, X., Verma, R.M.: In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* Association for Computing Machinery, New York, NY, USA, (2022), ASIA CCS '22, p. 1225–1227. https://doi.org/10.1145/3488932.3527288
39. Cao, S., Sun, X., Bo, L., Wu, R., Li, B., Tao, C.: In *Proceedings of the 44th International Conference on Software Engineering* Association for Computing Machinery, New York, NY, USA, (2022), ICSE '22, p. 1456–1468. https://doi.org/10.1145/3510003.3510219
40. Chen, Y., Ding, Z., Alowain, L., Chen, X., Wagner, D.: In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses* Association for Computing Machinery, New York, NY, USA, (2023), RAID '23, p. 654–668. https://doi.org/10.1145/3607199.3607242
41. Sun, S., Wang, S., Wang, X., Xing, Y., Zhang, E., Sun, K.: In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)* IEEE Computer Society, Los Alamitos, CA, USA, (2023), pp. 171–181. https://doi.org/10.1109/ICSME58846.2023.00027
42. Zhang, J., Liu, Z., Hu, X., Xia, X., Li, S.: Vulnerability detection by learning from syntax-based execution paths of code. IEEE Trans. Softw. Eng. **49**(8), 4196 (2023). https://doi.org/10.1109/TSE.2023.3286586
43. Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., Wagner, D., Ray, B., Chen, Y.: Vulnerability detection with code language models: how far are we? https://arxiv.org/pdf/2311.16169.pdf (2024). Accessed on June 11th, (2024)
44. Nguyen, S., Nguyen, T.T., Vu, T.T., Do, T.D., Ngo, K.T., Vo, H.D.: Code-centric learning-based just-in-time vulnerability detection. J. Syst. Softw. **214**, 112014 (2024). https://doi.org/10.1016/j.jss.2024.112014
45. Guo, Y., Hu, Q., Tang, Q., Traon, Y.L.: In *28th European Symposium on Research in Computer Security (ESORICS)* IEEE, (2023)
46. Andrade, R.O., Yoo, S.G.: Cognitive security: A comprehensive study of cognitive science in cybersecurity. J. Inf. Secur. Appl. **48**, 102352 (2019). https://doi.org/10.1016/j.jisa.2019.06.008
47. Jimenez, M., Rwemalika, R., Papadakis, M., Sarro, F., Le Traon, Y., Harman, M.: In *ESEC/FSE* ACM, New York, USA, (2019), p. 695–705. https://doi.org/10.1145/3338906.3338941
48. Garg, A., Degiovanni, R., Jimenez, M., Cordy, M., Papadakis, M., Le Traon, Y.: Learning from what we know: How to perform vulner-

ability prediction using noisy historical data. Empirical Softw. Eng. **27**(7), 169 (2022). https://doi.org/10.1007/s10664-022-10197-4
49. Denyer, D., Tranfield, D.: The Sage handbook of organizational research methods pp. 671–689 (2009)
50. Vom Brocke, J., Simons, A., Riemer, K., Niehaves, B., Plattfaut, R., Cleven, A.: Standing on the shoulders of giants: Challenges and recommendations of literature search in information systems research. Commun. Assoc. Inf. Syst. **37**(1), 9 (2015)
51. F-Droid Contributors. F-droid: free and open source android app repository. https://f-droid.org/ (2024). Accessed on January 30th, (2024)
52. Frederick, P.E.B., Boland Jr. E.: Computer (IEEE Computer) **45**(10) (2012). https://www.nist.gov/publications/juliet-11-cc-and-java-test-suite
53. Wang, S., Liu, W., Wu, J., Cao, L., Meng, Q., Kennedy, P.J.: In *2016 international joint conference on neural networks (IJCNN)* IEEE, (2016), pp. 4368–4374
54. Cui, Y., Jia, M., Lin, T.Y., Song, Y., Belongie, S.: In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019), pp. 9260–9269. https://doi.org/10.1109/CVPR.2019.00949
55. Lin, T.Y., Goyal, P., Girshick, R., He, K., Dollár, P.: A detection method for pavement cracks combining object detection and attention mechanism. IEEE Trans. Pattern Anal. Mach. Intell. **42**(2), 318 (2020). https://doi.org/10.1109/TPAMI.2018.2858826
56. Guo, Y.: Additonal materials. https://doi.org/10.6084/m9.figshare.25061420 (2024). Accessed on January 30th, (2024)
57. Alexopoulos, N., Brack, M., Wagner, J.P., Grube, T., Mühlhäuser, M.: In *USENIX Security* USENIX Association, Boston, MA, (2022), pp. 359–376. https://www.usenix.org/conference/usenixsecurity22/presentation/alexopoulos
58. Kupsch, J.A., Miller, B.P.: In *First International Workshop on Managing Insider Security Threats (MIST)* (2009), pp. 83–97
59. Guo, Y.: Software vulnerability detection datasets - function/methond level (2023)
60. National Institute of Standards and Technology. U.S. Department of Commerce. Nvd data feeds. https://nvd.nist.gov/vuln/data-feeds (2024). Accessed on January 30th, 2024
61. The MITRE Corporation. 2023 cwe top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html (2024). Accessed on January 30th, 2024
62. Dong, Z., Hu, Q., Guo, Y., Cordy, M., Papadakis, M., Zhang, Z., Traon, Y.L., Zhao, J.: In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* IEEE Computer Society, Los Alamitos, CA, USA, (2023), pp. 379–390. https://doi.org/10.1109/SANER56733.2023.00043
63. Yang, Z., Shi, J., He, J., Lo, D.: In *ICSE* ACM, New York, USA, (2022), p. 1482–1493. https://doi.org/10.1145/3510003.3510146
64. Karatas, G., Demir, O., Sahingoz, O.K.: Increasing the performance of machine learning-based IDSs on an imbalanced and up-to-date dataset. IEEE Access **8**, 32150 (2020). https://doi.org/10.1109/ACCESS.2020.2973219
65. Batista, G.E., Prati, R.C., Monard, M.C.: A study of the behavior of several methods for balancing machine learning training data. ACM SIGKDD Explor. Newslett. **6**(1), 20 (2004)
66. Nguyen, N., Nadi, S.: In *Proceedings of the 19th International Conference on Mining Software Repositories* Association for Computing Machinery, New York, NY, USA, (2022), MSR '22, pp. 1–5. https://doi.org/10.1145/3524842.3528470
67. Bommasani, R., Klyman, K., Longpre, S., Kapoor, S., Maslej, N., Xiong, B., Zhang, D., Liang, P.: The foundation model transparency index. https://arxiv.org/pdf/2310.12941.pdf (2023). Accessed on January 30th, (2024)
68. European Parliament. Artificial intelligence act. https://www.europarl.europa.eu/doceo/document/TA-9-2023-0236_EN.html (2023). Accessed on January 30th, 2024
69. Guo, Y., Bettaieb, S.: In *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* IEEE Computer Society, Los Alamitos, CA, USA, (2023), pp. 29–33. https://doi.org/10.1109/EuroSPW59978.2023.00008
70. Kekül, H., Ergen, B., Arslan, H.: Comparison and analysis of software vulnerability databases. Int. J. Eng. Manuf. **12**(4), 1 (2022). https://doi.org/10.5815/ijem.2022.04.01
71. Croft, R., Babar, M.A., Kholoosi, M.: In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* IEEE Computer Society, Los Alamitos, CA, USA, (2023), pp. 121–133. https://doi.org/10.1109/ICSE48619.2023.00022
72. Hanif, H., Md Nasir, M.H.N., Ab Razak, M.F., Firdaus, A., Anuar, N.B.: Journal of Network and Computer Applications **179**, 103009 (2021). https://doi.org/10.1016/j.jnca.2021.103009
73. Nie, X., Li, N., Wang, K., Wang, S., Luo, X., Wang, H.: In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* Association for Computing Machinery, New York, NY, USA, 2023, ISSTA (2023), p. 52–63. https://doi.org/10.1145/3597926.3598037