Check for updates

A Systematic Literature Review on Automated Software Vulnerability Detection Using Machine Learning

NIMA SHIRI HARZEVILI, Electrical Engineering & Computer Science, York University - Keele Campus, Toronto, Canada
ALVINE BOAYE BELLE, Electrical Engineering & Computer Science, York University, Toronto, Canada
JUNJIE WANG, Institute of Software Chinese Academy of Sciences, Beijing, China
SONG WANG, Electrical Engineering & Computer Science, York University, Toronto, Canada
ZHEN MING (JACK) JIANG, Electrical Engineering & Computer Science, York University, Toronto, Canada
NACHIAPPAN NAGAPPAN, Meta, Seattle, United States

In recent years, numerous Machine Learning (ML) models, including Deep Learning (DL) and classic ML models, have been developed to detect software vulnerabilities. However, there is a notable lack of comprehensive and systematic surveys that summarize, classify, and analyze the applications of these ML models in software vulnerability detection. This absence may lead to critical research areas being overlooked or underrepresented, resulting in a skewed understanding of the current state of the art in software vulnerability detection. To close this gap, we propose a comprehensive and systematic literature review that characterizes the different properties of ML-based software vulnerability detection systems using six major Research Questions (RQs).

Using a custom web scraper, our systematic approach involves extracting a set of studies from four widely used online digital libraries: ACM Digital Library, IEEE Xplore, ScienceDirect, and Google Scholar. We manually analyzed the extracted studies to filter out irrelevant work unrelated to software vulnerability detection, followed by creating taxonomies and addressing RQs. Our analysis indicates a significant upward trend in applying ML techniques for software vulnerability detection over the past few years, with many studies published in recent years. Prominent conference venues include the International Conference on Software Engineering (ICSE), the International Symposium on Software Reliability Engineering (ISSRE), the Mining Software Repositories (MSR) conference, and the ACM International Conference on the Foundations of Software Engineering (FSE), whereas *Information and Software Technology* (IST), *Computers & Security* (C&S), and *Journal of Systems and Software* (JSS) are the leading journal venues.

Our results reveal that 39.1% of the subject studies use hybrid sources, whereas 37.6% of the subject studies utilize benchmark data for software vulnerability detection. Code-based data are the most commonly used data type among subject studies, with source code being the predominant subtype. Graph-based and token-based input representations are the most popular techniques, accounting for 57.2% and 24.6% of the subject

Authors' Contact Information: Nima Shiri Harzevili, Electrical Engineering & Computer Science, York University - Keele Campus, Toronto, Ontario, Canada; e-mail: nshiri@yorku.ca; Alvine Boaye Belle, Electrical Engineering & Computer Science, York University, Toronto, Canada; e-mail: alvine.belle@lassonde.yorku.ca; Junjie Wang, Institute of Software Chinese Academy of Sciences, Beijing, China; e-mail: junjie@iscas.ac.cn; Song Wang, Electrical Engineering & Computer Science, York University, Toronto, Canada; e-mail: wangsong@yorku.ca; Zhen Ming (Jack) Jiang, Electrical Engineering & Computer Science, York University, Toronto, Ontario, Canada; e-mail: zmjiang@cse.yorku.ca; Nachiappan Nagappan, Meta, Seattle, Washington, United States; e-mail: nachiappan.nagappan@gmail.com.

ACM 0360-0300/2024/11-ART55
https://doi.org/10.1145/3699711

studies, respectively. Among the input embedding techniques, graph embedding and token vector embedding are the most frequently used techniques, accounting for 32.6% and 29.7% of the subject studies. Additionally, 88.4% of the subject studies use DL models, with recurrent neural networks and graph neural networks being the most popular subcategories, whereas only 7.2% use classic ML models. Among the vulnerability types covered by the subject studies, CWE-119, CWE-20, and CWE-190 are the most frequent ones. In terms of tools used for software vulnerability detection, Keras with TensorFlow backend and PyTorch libraries are the most frequently used model-building tools, accounting for 42 studies for each. In addition, Joern is the most popular tool used for code representation, accounting for 24 studies.

Finally, we summarize the challenges and future directions in the context of software vulnerability detection, providing valuable insights for researchers and practitioners in the field.

CCS Concepts: • **Security and privacy** → **Software security engineering**

Additional Key Words and Phrases: Source code, software security, software vulnerability detection, software bug detection, machine learning, deep learning

ACM Reference Format:

Nima Shiri Harzevili, Alvine Boaye Belle, Junjie Wang, Song Wang, Zhen Ming (Jack) Jiang, and Nachiappan Nagappan. 2024. A Systematic Literature Review on Automated Software Vulnerability Detection Using Machine Learning. *ACM Comput. Surv.* 57, 3, Article 55 (November 2024), 36 pages. https://doi.org/10.1145/3699711

1 Introduction

Automatic vulnerability identification is essential for ensuring software security [99]. Successes in the field of **Machine Learning (ML)** have inspired a lot of interest in using these models to find software vulnerabilities in general/traditional software systems [145]. ML models excel at detecting subtle patterns and correlations in large datasets [6]. They can automatically extract important features from raw data, such as source code, and detect hidden patterns that could reveal software defects. This capacity is critical in vulnerability detection, as vulnerabilities frequently entail subtle code characteristics and dependencies. In addition, ML models can handle a wide range of data types and formats, including source code [26], textual information [56], and numerical features such as commit characteristics [114]. They can use these data representations to effectively discover vulnerabilities. This versatility enables researchers to use a variety of data sources and include numerous features for comprehensive vulnerability detection.

Although many studies have used ML models to detect software vulnerabilities, there has not been a comprehensive and systematic review to consolidate the various approaches and characteristics of these techniques. Conducting such a systematic survey would be beneficial for practitioners and researchers in gaining a better understanding of the current state-of-the-art tools for vulnerability detection and could serve as an inspiration for future studies. This study conducts a comprehensive and detailed survey to review, analyze, describe, and classify software vulnerability detection studies from different perspectives. We analyzed 138 studies published in many software engineering flagship journals and conferences from January 2011 to June 2024. In this study, we investigated the following **Research Questions (RQs)**:

- *RQ1*: What is the trend of studies?
  - *RQ1.1*: What is the trend of studies over time?
  - *RQ1.2*: What is the distribution of publication venues?
- *RQ2*: What are the characteristics of software vulnerability detection datasets?
  - *RQ2.1*: What is the source of datasets?
  - *RQ2.2*: What are the most commonly used data types?
  - *RQ2.3*: What are the most commonly used input representations?

  - *RQ2.4*: What are the most commonly used embedding approaches?
- *RQ3*: What is the distribution of ML and **Deep Learning (DL)** models used for software vulnerability detection?
- *RQ4*: What are the most frequent types of vulnerabilities covered in the subject studies?
- *RQ5*: What are the most frequently used tools for software vulnerability detection?
- *RQ6*: What are possible challenges and open directions in software vulnerability detection?

This article makes the following contributions:

— We thoroughly analyze 138 studies that used ML models to detect security vulnerabilities regarding publication trends, distribution of publication venues, and types of contributions.
— We conduct a comprehensive analysis to understand the dataset, the processing of data, data representation, model architecture, tools, and types of covered vulnerabilities in the subject studies.
— We provide a classification of ML models used in vulnerability detection based on their architectures.
— We discuss distinct technical challenges of using ML techniques in vulnerability detection and outline key future directions.
— We share our results and analysis data as a replication package[1] to allow other researchers to easily follow this work and extend it.

We believe that this work is valuable for researchers and practitioners in software engineering and cybersecurity, especially those focused on software vulnerability detection and mitigation. It also benefits policymakers, software providers, and stakeholders interested in improving software security and reducing cyberattack risks, forming their software development, procurement, and risk management decisions.

The rest of the article is organized as follows. Section 2 provides background information and reviews related work. Section 3 outlines the research methodology proposed in this article. Section 4 addresses the RQs and presents the corresponding results. Section 5 discusses potential threats to the validity of this study. Finally, Section 6 presents the conclusion and suggests future directions.

2 Background and Related Work

In this section, we begin by defining vulnerability and outlining the key steps in detecting software vulnerabilities. We then review related surveys, emphasizing how they differ from our own.

2.1 Background

Software vulnerability management is crucial for ensuring software security and integrity [119]. With the increasing reliance on software for critical operations like financial transactions [39], vulnerabilities pose serious risks, including unauthorized access and service disruption. Effective management is essential for protecting user privacy, maintaining system availability, and ensuring trustworthiness. There are multiple steps in software vulnerability management, including vulnerability detection, vulnerability analysis, and vulnerability remediation. In the following subsections, we elaborate on each step in detail.

2.1.1 Vulnerability Detection. Vulnerability detection is critical in the overall process of managing software vulnerabilities [11]. It comprises detecting possible security weaknesses in software systems that attackers may exploit. There are several traditional techniques commonly used for vulnerability detection. In the *manual code auditing* method, human experts examine the source thoroughly to manually detect coding flaws, unsafe procedures, and possible vulnerabilities. *Static*

[1] https://github.com/dmc1778/CSURSurvey

analysis [35] involves using automated tools to analyze the source code or compiled binaries without executing the software under test. The goal of *dynamic analysis* [67, 102] is to evaluate the behavior of software while it is running. Running the software in a controlled environment or through automated tests while monitoring its execution and interactions with system resources is what it entails. However, dynamic analysis may have constraints in terms of significant system overhead [167]. One approach that falls under this category is the usage of fuzz testing for software vulnerability detection [42]. In fuzz testing, the input space for the program under test is identified, then the inputs are modified/mutated randomly or based on a set of already-defined rules to generate malformed inputs as well as boundary input values (i.e., edge cases). These tainted values are expected to hit parts of the program under test that are not properly validated, which results in serious security vulnerabilities like denial of service or remote code execution. *Hybrid code analysis* [25] is a strong approach that combines the benefits of static and dynamic analysis to increase the effectiveness of software vulnerability detection. Static analysis examines code without executing it. Its key strength is its ability to quickly scan the entire codebase and identify any flaws before the code executes. Yet, it often generates high false positives and has limited context on runtime behavior [52]. Dynamic analysis, however, involves running the code and monitoring its behavior in a real-time fashion. This method excels at finding runtime issues such as memory leaks.[2] Yet, the main drawback is that it is resource intensive, as you need to run the entire program under test to explore different code patches.

The hybrid model leverages the strengths of both approaches to ensure comprehensive coverage. Despite its benefits, implementing hybrid code analysis has technical complexities, such as integrating and synchronizing static and dynamic tools. Additionally, it demands significant computational resources and time, potentially slowing down development time.

2.1.2 Vulnerability Analysis. After the detection of vulnerabilities, the subsequent step in software vulnerability management is vulnerability analysis and assessment [130]. This step involves a further examination of identified vulnerabilities to assess their severity, impact, and potential exploitability. First, with regard to *severity*, accurately assessing software vulnerabilities is vital for several reasons. One reason is that it allows organizations to prioritize their response based on the severity of the vulnerabilities. Severity refers to the potential impact a vulnerability could have if exploited [15]. By accurately assessing the severity, organizations can focus their attention on high-severity vulnerabilities that pose significant threats to the security and functionality of the software system. Second, with regard to *impact*, accurately assessing vulnerabilities helps determine the potential impact they may have on the organization [43]. The term *impact* refers to the manifestations of exploiting a vulnerability, such as denial of service [53] or data breaches. By understanding the potential impact, organizations can make informed decisions regarding the urgency and priority of remediation efforts. Third, with regard to *exploitability*, accurate vulnerability assessment aids in understanding their potential exploitability [14]. This entails determining the possibility that an attacker will be successful in exploiting the vulnerability to infiltrate the software system.

2.1.3 Vulnerability Remediation. The process of resolving detected software vulnerabilities by different techniques such as patching, code modification, and repairing is referred to as software vulnerability remediation [59]. The fundamental goal of remediation is to eliminate or mitigate vulnerabilities to improve the security and dependability of the software system. One common approach to vulnerability remediation is applying patches provided by software vendors or open

[2] Please note that it is possible to detect memory leak vulnerabilities using static analysis techniques; however, application of dynamic analysis is more effective compared to static analysis.

source communities [156]. Patches are updates or fixes that address specific vulnerabilities or weaknesses identified in a software system.

2.1.4 ML for Software Vulnerability Detection. By utilizing data analysis, pattern recognition, and ML to find software security vulnerabilities, ML approaches have revolutionized software vulnerability detection [145]. These techniques improve the accuracy and efficiency of vulnerability detection, potentially allowing automated detection, faster analysis, and the identification of previously undisclosed vulnerabilities. One common application of ML in vulnerability detection is the classification of code snippets [27], software binaries, or code changes extracted from open source repositories such as GitHub or **Common Vulnerability and Exposure (CVE)**. ML models can be trained on labeled datasets, where each sample represents a known vulnerability or non-vulnerability. These models then learn to generalize from the provided examples and classify new instances based on the patterns they have learned. This method allows for automatic vulnerability discovery without the need for manual examination, considerably lowering the time and effort necessary for analysis.

ML models for detecting software vulnerabilities have promising advantages over traditional methodologies. Each benefit is discussed in detail in the next paragraph. *Automation* is a significant advantage. ML models can automatically scan and analyze large codebases, or system configurations, detecting potential vulnerabilities without requiring human intervention for each case [12]. This automation speeds up the detection process, allowing security teams to focus on verifying and mitigating vulnerabilities rather than manual analysis. With regard to *efficiency* and *scalability*, ML approaches offer faster analysis. Traditional vulnerability detection techniques rely on manual inspection or the application of pre-defined rules [128]. In contrast, ML approaches can evaluate enormous volumes of data in parallel and generate predictions quickly, dramatically shortening the time necessary to find vulnerabilities. With regard to *detection effectiveness*, ML models can uncover previously unknown vulnerabilities, commonly known as zero-day vulnerabilities [5]. These models may uncover signs of vulnerabilities even when they have not been specifically trained on them by learning patterns and generalizing from labeled data. This capability improves the overall security by helping to identify and address unknown weaknesses in software before they are exploited by attackers [1].

Figure 1 shows the overall pipeline of software vulnerability detection. The pipeline for software vulnerability detection using ML models involves several key stages.
The first stage is *data collection*, where data is gathered from various sources such as benchmark datasets including but not limited to the **National Vulnerability Database (NVD)** and the **National Institute of Standards and Technology (NIST) Software Assurance Reference Dataset (SARD)**, code repositories (GitHub), and specific open source projects (LibTIFF, FFMPEG). The *data preprocessing* stage involves tokenization, parsing (using tools like Joern,[3]) normalization, and feature extraction to convert raw code into analyzable formats. The *data representation* stage is where the preprocessed data is converted into appropriate representations, including graph-based representations such as control flow or dataflow graphs, token representations, or numerical attributes. In the *feature extraction* stage, once the data is represented in an appropriate form, these representations are converted into suitable features using different embedding techniques such as graph embedding or token vector embedding. In the *model inference stage*, appropriate DL models (e.g., **Recurrent Neural Networks (RNNs)**, **Graph Neural Networks (GNNs)**, Transformers, Autoencoders, and **Deep Belief Networks (DBNs)**), as well as traditional ML models (e.g., **Support Vector Machines (SVMs)**, Decision Trees, and Random Forests), are

[3]https://joern.io/

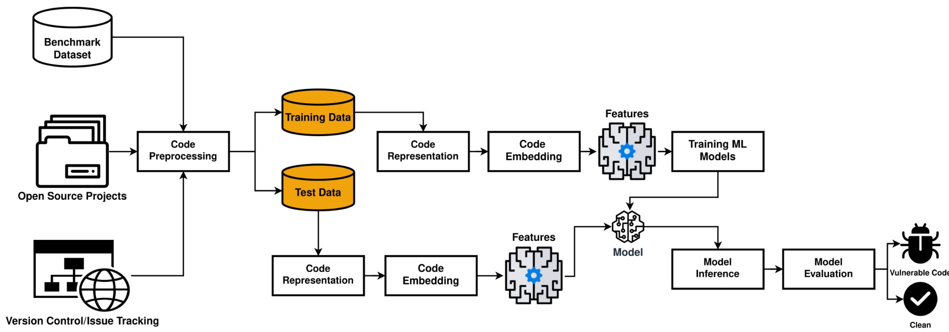

Fig. 1. Overall pipeline of software vulnerability detection.

Table 1. Comparison of Contributions between Our Survey and the Existing Related Surveys/Reviews

No.	Study	Data Source	Representation	Embedding	Models	Vulnerability Types	Tools
1	Le et al. [72]	✓	×	✓	✓	×	×
2	Ghaffarian & Shahriari [40]	✓	✓	✓	✓	×	×
3	Lin et al. [86]	✓	✓	✓	✓	×	×
4	Zeng et al. [173]	✓	✓	✓	✓	×	×
5	Semasaba et al. [124]	✓	✓	×	✓	✓	×
6	Sun et al. [133]	✓	×	×	×	✓	×
7	Kritikos et al. [69]	✓	×	×	×	✓	✓
8	Khan & Parkinson [66]	×	×	×	×	×	×
9	Nong et al. [112]	×	×	×	×	×	×
10	Chakraborty et al. [12]	×	×	×	×	×	×
11	Liu et al. [90]	×	×	×	×	×	×
12	Our survey	✓	✓	✓	✓	✓	✓

chosen based on the characteristics of the data. The training process includes splitting the data into training and test sets, feature engineering, hyperparameter tuning, and applying suitable training algorithms. In addition, *model evaluation* is often conducted using cross validation, performance metrics (i.e., accuracy, precision, and recall), confusion matrices, and ablation studies to ensure robust performance. This step ensures the models are accurate and reliable for detecting software vulnerabilities.

2.2 Related Work

There have been several existing survey papers on software vulnerabilities in the literature. In this section, we analyze the existing papers based on different aspects as shown in Table 1.

The table's columns represent different aspects of the surveys, such as the data source used, representation, feature embedding, ML models, vulnerability types, and tools employed for model building or dataset processing. *Data Source* indicates whether the survey reviewed vulnerability detection data sources. *Representation* discusses whether the survey considered source code representation in its analysis. *Embedding* checks whether the survey considered feature embedding. The table also considers the ML models in the sixth column. The table also checks whether the survey considers vulnerability types based on the **Common Weakness Enumeration (CWE)** number. The last column indicates whether the studies covered tools used for software vulnerability detection.

The works of Ghaffarian and Shahriari [40] and Kritikos et al. [69] are the closest surveys to ours when it comes to the detection of data-driven security vulnerabilities. In their surveys, they analyzed ML-based software vulnerability detection from various aspects as shown in Table 1. However, there are a couple of differences compared to our work. Specifically, our work surveys vulnerability detection from the following aspects: better understanding of attack patterns and tools used for software vulnerability detection. Understanding different types of vulnerabilities gives researchers insights into various attack patterns, enabling them to design detection techniques that can identify both known and unknown attack patterns. Understanding tools for software vulnerability detection reveals technological trends, helping researchers in this field leverage tools for reproducibility. It highlights the strengths and weaknesses of existing tools, guiding new developments. Popular tools offer community support, documentation, and shared knowledge, accelerating innovation and practical application of research.

Le et al. [72] reviewed data-driven vulnerability assessment and prioritization studies. They conducted a review of prior research on software assessment and prioritization that leverages ML and data mining methods. The major difference from ours is that we review software vulnerability detection techniques, which refers to the process of identifying potential vulnerabilities in software systems, whereas they survey assessment and prioritization techniques.

Lin et al. [86] examined the literature on using DL and neural network based techniques to detect software vulnerabilities. The major difference compared to our work is that we examine the trend analysis of papers published in software vulnerability detection in journal and conference papers because it provides a comprehensive understanding of the publishing patterns in a particular field or area of research. Trend analysis can shed light on the distribution of research output across various publication venues and the shifting preferences of researchers and authors.

Zeng et al. [173] discussed the growing focus on exploitable software vulnerabilities and the development of detection methods, especially using ML techniques. It reviews 22 recent studies employing DL for vulnerability detection and identifies four significant game-changers in the field. The survey compares these game-changers based on data sources, feature representation, DL models, and detection tools. Our survey differs in two key ways. First, we analyze publication trends in software vulnerability detection in journals and conferences, providing a comprehensive understanding of research trends. Second, we cover additional aspects beyond data sources, feature representation, and ML models including vulnerability types and detection tools.

Kritikos et al. [69] and Sun et al. [133] focused on cybersecurity and aimed to improve cyber resilience. Sun et al. [133] discussed the paradigm shift in understanding and protecting against cyber threats from reactive detection to proactive prediction, with an emphasis on new research on cybersecurity incident prediction systems that use many types of data sources. Kritikos et al. [69] discusses the challenges of migrating applications to the cloud and ensuring their security, with a focus on vulnerability management during the application lifecycle and the use of open source tools and databases to better secure applications. While both approaches aim to improve the security of applications, they differ in their focus and techniques used. They mainly focus on providing guidance and tools to support vulnerability management during the application lifecycle, whereas in our survey, we focus on software vulnerability detection using ML techniques on source code which aim at automating the identification of vulnerabilities in the source code or repository data (i.e., commit characteristics).

Khan and Parkinson [66] focused on vulnerability assessment, which is the process of finding and fixing vulnerabilities in a computer system before they can be exploited by hackers. This highlights the necessity for more studies into automated vulnerability mitigation strategies that can effectively secure software systems. However, vulnerability identification with ML approaches on source code entails analyzing a software's source code to spot security flaws. Instead of evaluating the safety of the entire system, this method concentrates on finding vulnerabilities in the code itself.

Nong et al. [112] explored the open science aspects of studies on software vulnerability detection and argued that there is a dearth of research on problems of open science in software engineering, particularly about software vulnerability detection. The authors conducted an exhaustive literature study and identified 55 relevant studies that propose DL-based vulnerability detection approaches. They investigated open science aspects including availability, executability, reproducibility, and replicability. The study revealed that 25.5% of the examined approaches provide open source tools.

Chakraborty et al. [12] investigated the performance of cutting-edge DL-based vulnerability prediction approaches in real-world vulnerability prediction scenarios. They find that the performance of the state-of-the-art DL-based techniques drops by more than 50% in real-world scenarios. The significant difference compared to our survey study is that in our work, we focus on the usage of ML models for software vulnerability detection and characterize the different stages in the pipeline of vulnerability detection. However, they focus on issues related to the use of state-of-the-art DL models for software vulnerability detection.

Liu et al. [90] discussed the increasing popularity of DL techniques in software engineering research due to their ability to address software engineering challenges without extensive manual feature engineering. The major difference compared to our study is that we focus on the usage of ML techniques in software vulnerability detection pipelines, whereas they emphasize replicability and reproducibility of the results reported in software engineering research studies.

3 Methodology

3.1 Sources of Information

In this article, we conducta systematic survey following other works [65, 116] to collect and examine studies from January 2011 to June 2024 focusing on software vulnerability detection using ML techniques. The overall workflow of our systematic approach is depicted in Figure 2. We target a set of popular and widely used digital libraries as the source of our data, including ACM Digital Library, ScienceDirect, IEEE Xplore, and Google Scholar. We developed a web crawler[4] based on Selenium[5] and Beautiful Soup[6] libraries. The reason we developed a web crawler is that it offers a reliable, scalable, and effective method for collecting relevant information from the web, which is very useful for academic research, specifically systematic literature review.

The period between January 2011 to June 2024 is an appropriate time interval for extracting software vulnerability detection studies for several reasons. One reason is the *increase in the volume and diversity of software vulnerabilities*. During the past decade, there has been a significant increase in the number and diversity of software vulnerabilities that have been discovered and reported.[7] As of 2021, there were 150,000 CVE records in NVD.[8] This increase has created a need for more sophisticated and effective methods for vulnerability detection, which has led to the development of new data-driven techniques. A second reason is *advancements in ML and data analytics*. The past decade has seen significant advancements in ML, including the development of DL algorithms [44, 55], natural language processing techniques [89], and other data-driven approaches that are highly effective in detecting software vulnerabilities.

3.2 Search Terms

Following existing surveys [72, 86, 124, 173], we devised the following search terms:

[4]https://github.com/dmc1778/CSURSurvey
[5]https://pypi.org/project/selenium/
[6]https://pypi.org/project/beautifulsoup4/
[7]https://nvd.nist.gov/general/news
[8]https://nvd.nist.gov/general/brief-history

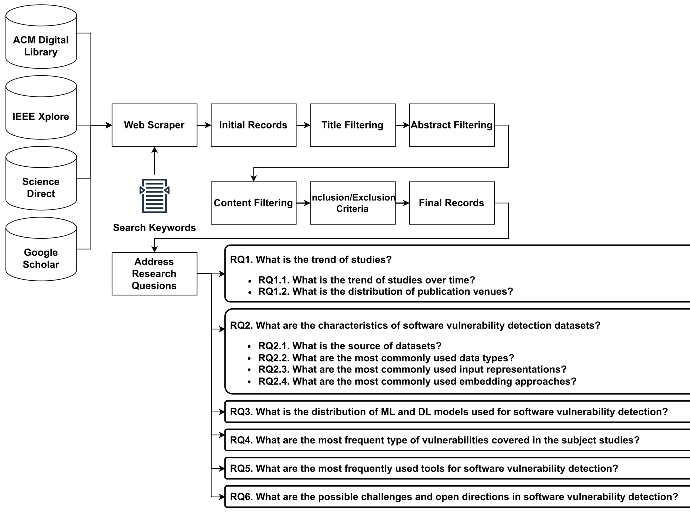

Fig. 2. Overall workflow of our systematic survey.

"*vulnerability detection*" OR "*Deep Transfer Learning Vulnerability Detection*" OR "*Transfer Learning Software Vulnerability Detection*" OR "*Transfer Learning Software Bug Detection*" OR "*Software Vulnerability Detection*" OR "*Vulnerability Detection Using Deep Learning*" OR "*Source Code Security Bug Prediction*" OR "*Source Code Vulnerability Detection*" OR "*Source Code Bug Detection*" OR "*Vulnerability Detection on Source Code Using Deep Learning*"

Using the keywords and our web scraper, we collected more than 15K initial records[9] from the subject digital libraries shown in Figure 2. After extracting initial records, we started the manual analysis and filtering of initial records in three stages including verification based on paper titles, abstracts, and contents. These three stages are explained in detail in the following subsections.

3.3 Study Selection and Quality Assessment

The process of selecting studies to be included in our survey involves the following stages: (1) initially choosing studies based on their title, (2) selecting studies after reviewing their abstracts, and (3) making further selections after reading the full articles. Note that the initial search results contain entries that are not related to software vulnerability detection. This might be caused by accidental keyword matching. We manually checked each paper and removed these irrelevant papers to ensure the quality of our survey dataset. We also observe that there exist duplicate papers among search results since the same study could be indexed by multiple databases. We then discarded duplicate studies manually.

The inclusion criteria are as follows: (1) the studies should have been peer reviewed (i.e., we do not include arXiv papers), (2) the studies should have experimental results, (3) the studies should propose a novel ML technique, (4) the studies should improve existing data-drive vulnerability detection techniques, and (5) the input to ML models should be either source code, text, commit, byte-code, or a combination of them. In addition, we have the following exclusion criteria to filter out irrelevant papers: (1) studies focusing on other engineering domains (electrical engineering, mechanical engineering, aerospace engineering, etc.), (2) studies addressing static analysis,

[9] https://github.com/dmc1778/CSURSurvey

dynamic analysis, hybrid analysis, and mutation testing, (3) review or survey studies, (4) studies focusing on vulnerability detection of web and Android applications, (5) studies belonging to one of the following categories: books, chapters, tutorials, or technical reports, and (6) studies focusing on malware detection on mobile devices, intrusion detection, and bug detection using static code attributes (i.e., Cyclomatic Complexities).

3.3.1 Title Filtering Stage. In this stage, we filter studies based on their titles. Since titles do not convey much information about the subject study, we only focused on *relevance to the initial keywords*. In this stage, we answer the following question: Do the titles contain specific keywords or phrases that are central to software vulnerability detection? For example, in the study titled "Toward Hardware-Based IP Vulnerability Detection and Post-Deployment Patching in Systems-on-Chip,"although the title includes our devised keyword *vulnerability detection*, the context indicates that the focus is on hardware and systems-on-chip rather than software engineering.

After the manual analysis on approximately 15K records, we collected 398 unique studies for further evaluation.

Abstract Filtering Stage. Given the list of studies filtered from the previous stage, we thoroughly analyzed the abstract of the studies. We decomposed the abstract of each paper into four major sections, including *Context*, *Objective*, *Approach*, and *Results/Findings*, as abstracts of research papers often follow such structure.

In this stage of filtering, we get 202 unique papers for further verification.

Content Filtering Stage. In this section, we analyze the content of each study in detail to perform the filtering process. Since there is more detail in the actual content of each study, we devise a set of criteria questions. We rely on the answers to these questions to assess the quality of the papers. If the answers to these questions are positive, the study is relevant; otherwise, we remove the paper from further examination. The questions are as follows: (1) Is there a clearly stated research goal related to software vulnerability detection in the introduction of the paper?; (2) Does the proposed vulnerability detection approach use ML or DL techniques?; (3) Is there a defined and repeatable technique?; (4) Is there any explicit contribution to software vulnerability detection?; (5) Is there a clear methodology for validating the technique?; (6) Are the subject projects selected for validation suitable for the research goals?; (7) Are the employed datasets relevant to software vulnerability detection?; (8) Are the type of input data to DL and ML models relevant to software vulnerability detection? (valid data types include source code, binary code, text, and commit metrics); (9) Are there control techniques or baselines to demonstrate the effectiveness of the software vulnerability detection technique?; (10) Are the evaluation metrics relevant (e.g., evaluate the effectiveness of the proposed technique) to the research objectives?; and (11) Do the results presented in the study align with the research objectives, and are they presented in a clear and relevant manner?

The filtering process in this stage resulted in 138 subject studies to address the RQs. We used these 138 studies to create taxonomies which are explained in detail in the next section.

3.4 Taxonomy Development and Classification Methodology

In this section, we present the methodology used to develop our taxonomy and classify the selected papers based on our RQs. The process is done in an incremental approach following existing studies [53]. The foundation of our taxonomy is anchored in a systematic analysis of the literature, guided by the specific RQs designed to explore various dimensions of software vulnerability detection. Each RQ serves as a focal point for our classification, ensuring a structured and coherent approach.

Extraction of Relevant Information. We meticulously examined each selected paper to extract relevant text segments related to the RQs. For RQ2, which pertains to the sources of datasets, we examine the experiential setup sections of each study. This section is the most commonly used section where authors discuss the source of datasets.[10] This allows us to understand the types of datasets the authors used to evaluate their proposed software vulnerability detection techniques. For RQ3, we analyzed the section detailing the proposed approach for software vulnerability detection. This involved identifying descriptions of the employed ML and DL models. One of the main sources of information that clearly explains the proposed approach is the overall architecture, which depicts the entire process of the proposed technique. For RQ4, we examine the vulnerability types covered in the subject study. These types often use the CWE system, which is easy to locate in the paper. We search for any keywords that start with *CWE* in the paper. If we find any CWE IDs mentioned, we record them. Otherwise, we note that the paper does not specify which vulnerability types their study aims to detect. Please note that some papers do not mention the CWE ID. For instance, for Integer Overflow (CWE-190), they only use the original title instead of the CWE ID. Therefore, we search for both CWE IDs and other related vulnerability keywords. For RQ5, we thoroughly analyzed the experimental sections, particularly the implementation sections of the subject studies to extract information about the tools used for building the ML models. Our empirical evaluation revealed that the authors usually use the keywords *implementation* or *built* to describe the tools they used. For RQ6, we examine the introduction section of the subject study, as authors often explicitly mention the specific problem they address in software vulnerability detection.

Create Preliminary Taxonomies. Initially, we establish a preliminary taxonomy that groups the studies based on defined RQs, which provides a basic framework for organizing the studies in a meaningful and systematic manner. For example, for the first study, we create preliminary taxonomies for RQ1 through RQ6. After thoroughly addressing all RQs for a given study, we move on to the next study.

Iterative Refinement. Once the initial taxonomy is created, we proceed to expand and refine it as we delve deeper into the analysis of each RQ across all subject studies. The authors then expand the taxonomy by assigning new papers to the preliminary taxonomy. If a new paper cannot fit into any of the existing categories within the taxonomy, a new category is created that reflects the unique characteristics of that paper. To ensure the accuracy of the taxonomy, the second and third authors (who are not involved in the taxonomy creation process) randomly select 20 papers from the workflow and check the created taxonomies for any discrepancies. After identifying any disagreements, they proceed to mark them. Subsequently, all authors engage in discussions to address and resolve these disagreements. Initially, the disagreement rate was 30%, but after a second round of review and cross checking of the papers, we were able to eliminate all disagreements.

Resolving Disagreements. During the extraction process, if we encountered conflicting information or interpretations, we collaboratively discussed these discrepancies to reach a consensus. This collaborative effort ensured that our classification remained consistent and accurate. By following this rigorous methodology, we ensured that our taxonomy is grounded in detailed and systematic analyses of the literature. This approach provides a clear and coherent framework for classifying the selected papers and addressing each RQ comprehensively.

[10]Please note that if we could not find the dataset name and source in the experimental setup section, we looked for other sections.

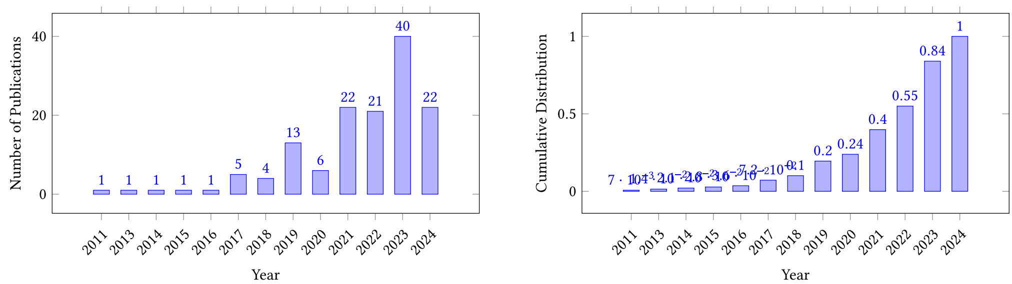

Fig. 3. Publication trend of vulnerability detection studies.

4 Results

In this section, we present our analyses and findings to address the RQs.

4.1 RQ1: What Is the Trend of Studies?

To understand the trend of publications, we examined the publication dates and the venues in which they were presented.

4.1.1 RQ1.1: What Is the Trend of Studies over Time? Figure 3 demonstrates the publication trend of software vulnerability detection studies published over 13 years (i.e., between January 2011 and June 2024). It is observable that the number of publications has gradually increased over the years.

We also analyze the cumulative number of publications shown in Figure 3. It is noticeable that the curve fitting the distribution shows a significant increase in slope between 2020 and 2024, suggesting that the usage of ML techniques for software vulnerability detection has become a prevalent trend since 2020.

4.1.2 RQ1.2: What Is the Distribution of Publication Venues? In this study, in general, we studied and reviewed 138 studies from various publication venues, including 61 studies from conferences and symposiums and 77 studies from journals. Table 2 shows the distribution of studies for each publication venue. A total of 44.2% of the publications are published in conferences and symposiums, whereas 55.7% of the studies have been published as articles in journals. It is observable that ICSE, ISSRE, MSR, and FSE are the most popular venues that have the highest number of studies. Meanwhile, among the journal venues, IST, C&S, and JSS have the highest number of studies—that is, 13, 12, and 12 studies, respectively.

Answer to RQ1

(1) The application of ML techniques for software vulnerability detection has had a remarkable rising trend in the past few years.
(2) Many papers are published in the past 4 years (i.e., 2021, 2022, 2023, and 2024).
(3) ICSE, ISSRE, MSR, and FSE are the most popular conference venues. In terms of journal venues, IST, C&S, and JSS are the most popular ones.

4.2 RQ2: What Are the Characteristics of Software Vulnerability Detection Datasets?

In this section, we examine data used in vulnerability detection studies and conduct a comprehensive analysis of the steps of data source, data type, and data representation.

Table 2. Distribution of Publications Based on Conference and Journal Venues

Conference Venue	# Studies	References	Journal Venue	# Studies	References
ICSE	9	[11, 113, 129, 135, 146, 147, 150, 155, 170]	IST	13	[9, 10, 17, 30, 31, 108, 126, 127, 139, 149, 158, 175, 181]
ISSRE	6	[153, 169, 172, 179, 180, 185]	C&S	12	[36, 45, 47, 63, 68, 77, 131, 132, 138, 148, 152, 164]
MSR	5	[19, 38, 54, 56, 105]	JSS	12	[7, 8, 13, 16, 32, 91, 98, 106, 114, 136, 143, 171]
FSE	5	[78, 81, 103, 111, 184]	TDSC	6	[83, 84, 87, 95, 188, 189]
IJCAI	4	[23, 34, 96, 187]	TSE	5	[26, 82, 122, 151, 174]
ASE	3	[73, 110, 177]	TIFS	4	[58, 142, 154, 157]
NDSS	2	[85, 125]	ISA	4	[134, 159, 176, 182]
NeurIPS	2	[3, 183]	TOSEM	3	[20, 109, 190]
TrustCom	2	[93, 165]	TKDE	2	[76, 97]
OOPSLA	2	[79, 118]	IS	2	[41, 64]
CCS	2	[115, 163]	ESA	2	[92, 140]
ICLR	2	[28, 71]	CN	1	[178]
QRS	2	[74, 141]	TFS	1	[94]
USENIX	1	[162]	SQJ	1	[33]
MASCOTS	1	[37]	PL	1	[80]
KDDM	1	[107]	P&S	1	[75]
ISSTA	1	[22]	Nature	1	[60]
IJCNN	1	[48]	KBS	1	[186]
ICTAI	1	[117]	FGCS	1	[49]
ICECCS	1	[21]	EAAI	1	[144]
ICBD	1	[168]	CEE	1	[120]
GLOBCOM	1	[166]	BRA	1	[4]
DSAA	1	[104]	ASC	1	[57]
CDSN	1	[137]			
CARS	1	[70]			
SANER	1	[29]			
ENTCC	1	[62]			
MCSoC	1	[46]			
Overall	**61**			**77**	

4.2.1 RQ2.1: What Is the Source of Datasets? One of the main challenges in ML-based software vulnerability detection is the insufficient amount of data available for model training [19, 88]. Consequently, there exists a gap in research on how to obtain sufficient datasets to facilitate the training of ML models for software vulnerability detection. To this end, we analyze the sources of datasets in the subject studies. Our analysis reveals that datasets for this purpose can be broadly classified into four categories: *Benchmark*, *Hybrid*, *Open Source Software*, and *Repository* sources. Among the subject studies, 39.1% of them use *Hybrid* as the data source for the detection of software vulnerability. They use a combination of various sources of data, such as benchmarks, repositories, and open source projects, to provide a comprehensive and multi-faceted resource for software vulnerability detection [36, 141]. These datasets combine the benefits of each data source to provide richer and more diversified information, which is critical for building and verifying robust vulnerability detection systems. *Benchmark* datasets used by 37.6% of the subject studies play a crucial role in the field of software vulnerability detection by providing standardized, high-quality data that researchers can use to evaluate and compare the effectiveness of their detection technique [127, 159]. Using benchmark datasets facilitates the construction of ML models for software vulnerability detection. However, they may not include zero-day vulnerabilities, which have a significant impact. Among the subject studies, 13.7% of them collect datasets from online repositories which we classify as the *Repository* category. These datasets are gathered from publicly available projects hosted on repository websites such as GitHub or Stack Overflow [28, 118, 184]. These repositories hold a plethora of data, including source code, commit history, issue trackers, and documentation. Repositories keep detailed records of any changes made to a codebase, such as commit messages, diffs, and timestamps [101]. This comprehensive history enables researchers to trace the lifecycle of vulnerabilities from introduction to resolution (please refer to the work of Iannone et al. [61]). The fourth source is open source software, accounting for 9.4% of the subject studies, which provides a rich and diverse source of data for software vulnerability detection [126, 163]. These projects are publicly accessible and typically have a large community of contributors who continuously update and maintain the code. Some example open source projects include but are not limited to

Table 3. Detailed Distribution of Benchmark Sources

No.	Source	# Studies	References
1	SARD	33	[9, 11, 20, 21, 31, 34, 36, 37, 47, 49, 63, 68, 83–85, 87, 95, 137–143, 148, 152, 155, 157, 158, 165, 179, 180, 189]
2	NVD	32	[10, 11, 19, 30–32, 36, 37, 49, 54, 63, 68, 70, 73, 83–85, 94, 95, 113, 132, 137, 142, 143, 155, 157–159, 174, 179, 180, 189]
3	Smartbugs Wild	12	[7, 13, 57, 91, 103–105, 134, 153, 169, 177, 185]
4	Big-Vul	8	[32, 38, 81, 98, 108, 110, 129, 188]
5	Reveal	6	[78, 98, 140, 150, 151, 174]
6	Juliet Test Suit	5	[23, 29, 75, 148, 164]
7	ESC	5	[76, 96, 97, 169, 187]
8	D2A	5	[22, 29, 127, 140, 174]
9	SolidiFi-benchmark	5	[7, 103–105, 134]
10	Fan et al.	4	[22, 78, 150, 151]
11	Vuldeepecker	4	[16, 17, 140, 190]
12	VSC	4	[76, 96, 97, 187]
13	NDSS	3	[71, 75, 107]
14	PROMISE	3	[74, 146, 172]
15	FUNDED	2	[60, 174]
16	F-Droid	2	[26, 122]
17	Android/iOS	2	[26, 122]
18	SySeVr	2	[17, 190]
20	Others	25	[7, 32, 33, 41, 46, 48, 57, 60, 64, 70, 73, 81, 82, 95, 129, 134–136, 140, 142, 164, 166, 174, 181, 182]
–	**Unique Total**	**99**	–

FFmpeg, *QEMU*, *OpenSSH*, and *LibTIFF*. The open nature of these projects means that they are often inspected carefully by numerous developers, which can lead to the discovery and documentation of various vulnerabilities.

Table 3 shows the detailed distribution of benchmark data used in the subject studies. As it is observable, *SARD* and *NVD* are the most widely used sources of data in the *Benchmark* category. SARD is a comprehensive set of test cases created exclusively for testing software systems. It was developed by NIST[11] as part of their efforts to improve the quality and safety of software systems. SARD offers a wide range of synthetic and real-world test scenarios intended to reflect many sorts of software vulnerabilities. Another major source of benchmark data is NVD, which is a comprehensive repository of publicly disclosed software vulnerabilities. NVD entries are based on the CVE system, which provides standardized identifiers and descriptions for each vulnerability. CVEs are assigned by CVE Numbering Authorities[12] and are a cornerstone of NVD. Each entry in NVD includes detailed information about the vulnerability, such as its description, severity (using the Common Vulnerability Scoring System), impacted software versions, references to related advisories, and mitigation advice. Smartbugs Wild[13] is also the third most commonly used (accounting for 12 studies) dataset for software vulnerability detection within the field of smart contracts. Smartbugs Wild contains more than 47K smart contracts mined from the main network of Ethereum, which includes a wide variety of real-world smart contracts, providing a useful dataset for testing and assessing vulnerability detection techniques. Please note that the key factor confirming the validity of a benchmark dataset is its continuous updating. As the nature of vulnerabilities evolves and more zero-day vulnerabilities emerge, these datasets need to be updated to reflect the latest software vulnerability patterns. This is why researchers do not rely solely on benchmark data for building ML models.

Table 4 shows the detailed distribution of the *Repository* source of data. As shown, *GitHub* is the most popular source of data for software vulnerability detection, accounting for 27 subject studies. One benefit of utilizing GitHub as a data source is that it gives you access to real-world code written by developers, which can be used to train and test vulnerability detection models. The

[11] https://www.nist.gov/
[12] https://www.cve.org/ProgramOrganization/CNAs
[13] https://github.com/smartbugs/smartbugs-wild

Table 4. Detailed Distribution of Repositories Used for Collecting Data

No.	Source	# Studies	References
1	GitHub	27	[10, 11, 19, 20, 28, 48, 54, 73, 79, 80, 93, 94, 106, 111, 113–115, 118, 120, 132, 142, 147, 149, 159, 175, 176, 184]
2	CVE	20	[9, 11, 19, 38, 47, 58, 60, 75, 87, 94, 113, 115, 131, 132, 141, 147, 152, 154, 174, 176]
3	Etherscan	13	[4, 7, 8, 58, 62, 82, 120, 125, 135, 168, 171, 176, 178]
4	Bugzilla	4	[19, 114, 166, 184]
5	Jira	3	[19, 80, 184]
6	PyPI	1	[3]
–	**Unique Total**	**51**	–

Table 5. Detailed Data Types Used in the Subject Studies

Category	Data Type	# Studies	Total	References
Code based	Source code	108	128	[3, 7–11, 13, 16, 20–23, 26, 28, 29, 31, 32, 34, 36–38, 41, 47–49, 54, 60, 63, 64, 68, 70, 73, 74, 77–85, 87, 91–98, 103, 104, 106, 108–110, 113, 118, 120, 122, 126, 127, 129, 131, 132, 134–138, 140–144, 146, 147, 149–154, 157–159, 162, 163, 165, 169–172, 174–177, 179–181, 183, 185–190]
	Binary code	18		[4, 45, 46, 57, 58, 62, 71, 75, 105, 107, 117, 125, 139, 148, 164, 168, 178, 182]
	Image	2		[76, 155]
Hybrid	–	4	4	[17, 19, 30, 56]
Commit Metrics	–	4	4	[111, 114, 115, 166]
Text	–	2	2	[33, 184]
Unique Total	–	–	138	–

second commonly used source of repository data is the *CVE* system, which is a widely recognized and utilized framework for identifying, cataloging, and referencing publicly disclosed vulnerabilities. Each vulnerability in the CVE system is given a unique identification known as a CVE ID (e.g., CVE-2023-33976). This standardized identifier facilitates easy reference and communication across various platforms and tools. CVE entries provide detailed descriptions of vulnerabilities, outlining the nature of the issue, the affected software, and the potential impacts. The third commonly used source of repository data is Etherscan,[14] a popular blockchain explorer for the Ethereum blockchain. Etherscan provides users with extensive information about Ethereum transactions, addresses, tokens, and smart contracts. It offers detailed insights into deployed smart contracts, including the contract's source code (if verified), transactions, and execution history. Users can access the complete history of transactions involving a smart contract, with details about function calls, input parameters, and transaction results.

4.2.2 RQ2.2: What Are the Most Commonly Used Data Types? When it comes to detecting software vulnerabilities, datasets can have varying data types. Existing software vulnerability detection models, for example, can find vulnerabilities in source code or commits. It is crucial to carefully examine these data types, as they require different preprocessing techniques and must be represented differently when using ML models. Additionally, distinct data types necessitate different architectural approaches for ML models. This section provides an overview of the various data types and their distributions. We classified the data types of the employed datasets into four broad categories: *Code*, *Text*, *Numerical*, and *Hybrid*.

The majority of the subject studies (92.7%) primarily focus on analyzing source code for software vulnerability detection, underscoring the importance of code-level analysis in identifying vulnerabilities. Repository-level data, such as textual reports and logs, account for 1.4%, whereas commit characteristics (numerical data) account for 2.8%. Additionally, 2.8% of the studies adopt a hybrid approach, combining both code-level analysis and repository-level data.

Table 5 elaborates on the detailed data type categories used in the subject studies. The table shows that 128 subject studies used a code-based category and the major data type of this category is *Source code* [34, 179]. *Binary code* is the second major data type in the code-based category [58, 117], accounting for 18 subject studies.

[14]https://etherscan.io/

4.2.3 RQ2.3: What Are the Most Commonly Used Input Representations? As noted in earlier sections, research studies focusing on software vulnerability detection rely on diverse sources of data and data types. This variability urges the adoption of various representation strategies, architectural approaches, and design assumptions for ML models.

We classified the input representation of employed datasets into five broad categories: *Graph*, *Token*, *Tree*, *Commit Metrics*, and *Hybrid*. The most popular input representation is the use of *Graph*, accounting for 57.2% of the subject studies. *Token* follows closely, representing a substantial portion (24.6%) of the subject studies. *Tree* representation is the third most common approach, accounting for 11.5% of the subject studies. The *Commit Metrics* and *Hybrid* categories have the smallest portion, accounting for 2.8% and 2.1% of the subject studies, respectively. In the following paragraphs, we elaborate on each category in detail.

Graph/Tree-Based Representation [63, 126]. This type allows for the detection of complex patterns and relationships between different code elements. By representing source code as a graph or tree, we can capture not only the syntax and structure of the code but also its semantics, control flow, and dataflow. There are many graph/tree-based representation techniques, such as AST (Abstract Syntax Trees) [100, 161] and CPG (Code Property Graph) [34, 41, 183] used to transform source code into AST and CPG representations.

Token-Based Representation [45, 140]. This typetreats the source code as string token sequences and then transforms source code into token vectors. The input data is first broken down into tokens, which are then turned into numerical vectors that can be processed by ML algorithms. Tokenization involves breaking down a string of text or source code into smaller units, or tokens, which can then be used as the basis for further analysis. In the case of source code, tokens might include keywords, operators, variables, and other elements of the programming language syntax.

Commit Metrics [114, 115]. This type leverages the metrics extracted from commits to represent code commits. Commit-level features, such as the number of code changes, the number of modified lines, and the programming language used, can be used to train ML models. These models may then learn patterns and connections between commit attributes and the presence of vulnerabilities, allowing for automatic detection of new commits.

Hybrid Representation [19, 30]. This type employs a variety of representations to discover software security vulnerabilities. Combining diverse representations of input data can result in a more comprehensive and richer input representation of source code, which can help vulnerability detection techniques perform better in tasks like prediction and detection.

Table 6 shows the representation techniques distributed by different artifacts used by ML models. It is evident that *Graph/Tree-based representation* is the most prevalent technique, with a total of 96 studies employing this method. These studies represent the input to ML models using various forms: *Source code as a graph*, *Source code as a tree*, *Binary code as a graph*, and *Binary code as a tree*. Notably, *Source code as a graph* is the predominant representation technique, used by 71 studies. Furthermore, 33 subject studies employed *Token-based representation*. Among them, 23 studies represented source code as a token sequence, 9 studies modeled binary code as tokens, and 2 studies represented text as token sequences.

Figure 4 shows the distribution of data type representation in software vulnerability detection studies over time. As shown in the figure, *Graph-based* representation shows a substantial presence compared to other input representation techniques. There are a couple of reasons for this trend. First, graphs provide a natural and intuitive way to represent the structural relationships within the source code. By modeling the code as a graph, the relationships between functions, classes, methods, and variables can be captured effectively. *Token-based* representation has also gained popularity, with a peak occurrence in 2023. This is because it provides a fine-grained representation

Table 6. Distribution of Input Representations in the Subject Studies

Category	Artifact	# Studies	Total	References
Graph/Tree	Source code as a graph	71	96	[3, 7–11, 13, 20, 21, 29, 31, 34, 36, 38, 41, 47, 49, 60, 63, 64, 68, 70, 77–79, 81, 82, 84, 85, 91–93, 95–97, 103, 104, 106, 110, 120, 126, 127, 129, 132, 134, 136, 138, 141, 143, 144, 147, 150–154, 157, 158, 165, 170, 172, 174, 175, 179–181, 183, 185, 187–189]
	Source code as a tree	15		[22, 28, 32, 74, 80, 83, 87, 94, 98, 142, 146, 159, 176, 177, 186]
	Binary code as graph	8		[46, 58, 75, 105, 117, 139, 148, 178]
	Binary code as tree	1		[4]
Token	Source code as a token	23	33	[16, 23, 26, 37, 48, 54, 56, 73, 108, 109, 113, 118, 122, 131, 135, 137, 140, 149, 162, 163, 169, 171, 190]
	Binary code as a token	9		[45, 57, 62, 71, 107, 125, 164, 168, 182]
	Text as a token	2		[33, 184]
Commit Metrics	–	4	4	[111, 114, 115, 166]
Hybrid	–	3	3	[17, 19, 30]
Image	–	2	2	[76, 155]
Unique Total	–	–	**138**	

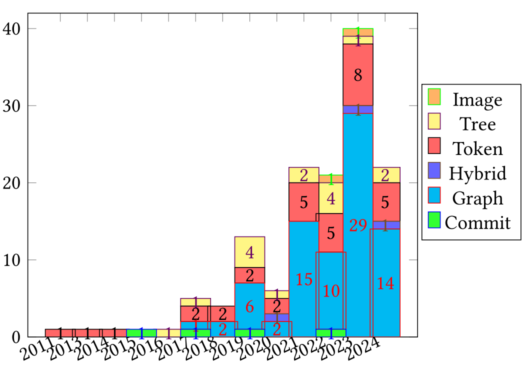

Fig. 4. Distribution of data type representations in software vulnerability detection studies over time.

of the code. It simplifies the code analysis process by reducing the complexity of the code to a sequence of tokens, making it easier to apply ML models.

4.2.4 RQ2.4: What Are the Most Commonly Used Embedding Approaches? In this section, we look at embedding methods that can convert these representations explored in the previous section into inputs that ML models can understand. The representation approaches are in a human-readable format and cannot be directly understood by computers. As a result, researchers applied various embedding approaches to translate these representations into numerical format. We discuss the embedding techniques in the following paragraphs.

Graph Embedding (32.6%) [97, 117]. This is the most commonly used embedding technique among the subject studies, accounting for 32.6%, which is mostly used by graph neural networks for its capability to capture the structural relationships between different code components.

Token Vector Embedding (29.7%) [79, 190]. This is the second most popular technique used by subject studies, accounting for 29.7% of examined papers. In this technique, input is converted into a sequence of tokens and each token is transformed into a numeric value. Then, these values are fed into ML models for training operations.

Hybrid (16.6%) [19, 41]. We find that 16.6% of the subject studies use multiple embedding techniques to convert inputs to ML models. Different embedding techniques capture different aspects of the data. By combining multiple techniques, researchers can leverage the complementary

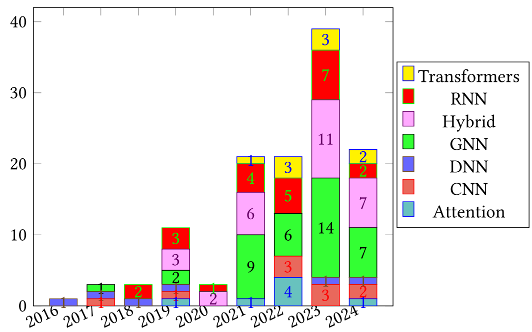

Fig. 5. Trend of DL models over time.

information provided by each technique. For example, some embedding techniques may focus on syntax, whereas others may capture semantic or contextual information.

Transformer Embedding (7.2%) [48, 153]. Transformer embedding is used in 7.2% of the subject studies. Despite its lower prevalence, the use of Transformers is notable because of their powerful capabilities in natural language processing, which can be adapted to analyze code.

Others (13.7%) [126, 146, 163]. The remaining 13.7% that seldom emerge and do not belong to any group are classified as *Others*.

Answer to RQ2

(1) A total of 37.6% of the subject studies use *benchmark* datasets for software vulnerability detection. This can be because benchmark datasets are readily accessible to all researchers and can facilitate the reproducibility of studies.
(2) The most common data type among the examined vulnerability detection studies is the *Code-based data type*. In this category, *Source code* is the most prominent sub-type.
(3) *Graph-based* and *Token-based* input representations are the most popular input representation techniques used by subject studies, accounting for 57.2% and 24.6% of the subject studies, respectively.

4.3 RQ3: What Is the Distribution of ML and DL Models Used for Software Vulnerability Detection?

In this section, we provide detailed information about the various ML models utilized for software vulnerability detection. Initially, we present an analysis of the usage distribution of models based on the subject studies. Subsequently, we investigate the distribution of the usage of specific DL models used in the subject studies over time. However, we have not extensively analyzed the distribution of classic ML models since their prevalence is relatively small compared to DL models. However, we provide a comprehensive list of classic ML models that have been commonly used in subject studies.

The majority of studies (88.4%) use DL models for software vulnerability detection [82, 127, 159], whereas only 7.2% of the studies use classic ML models [19, 107, 184]. Some of the subject studies also use a combination of DL and ML models, accounting for 1.4% of studies. The remaining (2.8%) are classified as Others.

The graph in Figure 5 illustrates the usage trend of DL models in detecting software vulnerabilities from 2016 to 2024. According to the trend, DL models were first introduced in 2016 for vulnerability detection, and since then, the use of RNNs for vulnerability detection has shown an upward trend. The graph also demonstrates a rising trend in using GNNs for vulnerability detection from 2021 to 2024. This can be because GNNs are more powerful than RNNs in detecting vulnerabilities, as they can capture more meaningful and semantic representations of input source code.

Table 7 shows the distribution of DL models used in the subject studies. As shown in the table, *Recurrent Models* are the most commonly used DL models for software vulnerability detection. In this category, BiLSTM is the most frequently used recurrent model, appearing in 20 studies. GRU and LSTM are also popular models with 14 and 13 studies, respectively. *Graph Models* are the second most widely used class of DL models for software vulnerability detection. It is observable that GCN is the most prevalent model, appearing in 22 studies. GNN, GGNN, and GAT are also commonly used, accounting for 13, 9, and 8 subject studies, respectively. The presence of these models highlights the importance of capturing graph structures and relationships between code elements in vulnerability detection. *Convolutional Models* are used in 19 studies. While not as prevalent as recurrent or graph models, CNNs are still considered effective for capturing local patterns and features in vulnerability detection tasks.

Table 8 shows the distribution of classic ML models used in subject studies. As shown in the table, Random Forest is the most frequently used ML model, appearing in seven studies. Naive Bayes, SVM, and KNN are popular choices, with 5, 4, and 4 occurrences, respectively. Random Forest is an ensemble learning method that builds multiple Decision Trees and merges their outputs to make a final prediction. This ensemble approach helps improve the robustness and accuracy of detection, making it effective for detecting software vulnerabilities. Naive Bayes is popular because it is computationally efficient and easy to implement. It requires less training data compared to more complex algorithms, making it faster in both training and prediction phases [2, 50, 51].

Table 7 also shows one study that uses n-gram models for software vulnerability detection. N-gram models serve an important role in capturing local context using word sequence probabilities. An n-gram model predicts the likelihood of a word based on the preceding n-1 words, successfully describing the local structure of the language [18, 123]. N-gram models are effective at identifying patterns within sequences of tokens (e.g., words, characters, or code elements). In the context of code, an n-gram model can be trained on large codebases to understand the typical sequences of code elements.

4.3.1 Comparison of ML Models with Manual Code Analysis. When it comes to software vulnerability detection, ML models are far superior to conventional manual code analysis techniques. ML-based software vulnerability detection facilitates efficiency and scalability by automating the analysis of massive amounts of code. This ability is essential in the current software development environment, where quick and comprehensive security evaluations are required due to complex systems and frequent changes. This efficiency lowers the possibility of human error that comes with manual inspections while simultaneously speeding up the detection process. Additionally, preemptive threat detection and ongoing monitoring are made easier by ML models. But even with these benefits, human code analysis is still essential for handling some crucial situations. The best people to handle special circumstances like zero-day vulnerabilities [5]—vulnerabilities when exploits are found and used before software developers have a chance to mitigate them—are human analysts.

4.3.2 Transfer Learning for Software Vulnerability Detection. Transfer learning is crucial for software vulnerability detection. First, high-quality labeled datasets for software vulnerability detection are often scarce and expensive to produce because labeling requires expert knowledge [19, 87,

Table 7. Distribution of DL Models in the Subject Studies

Category	Model Name	# Studies	Total	References
Recurrent Models	BiLSTM	20	65	[47, 57, 63, 64, 83, 85, 87, 113, 120, 131, 136, 139, 148, 159, 168, 176, 182, 185, 189, 190]
	GRU	14		[47, 54, 63, 73, 76–79, 125, 139, 142, 144, 147, 181]
	LSTM	13		[26, 28, 63, 82, 94, 95, 125, 139, 149, 152, 158, 181, 186]
	BGRU	10		[32, 63, 68, 75, 83, 84, 138, 139, 164, 190]
	TreeLSTM	3		[8, 78, 147]
	RNN	3		[37, 139, 154]
	BRNN	2		[109, 139]
Graph Models	GCN	22	63	[7, 8, 20, 21, 31, 41, 46, 75, 78, 81, 91, 97, 104, 110, 127, 136, 147, 150, 172, 179, 181, 187]
	GNN	13		[3, 8, 10, 11, 20, 28, 105, 106, 136, 143, 151, 152, 183]
	GGNN	9		[29, 31, 36, 77, 92, 129, 142, 154, 188]
	GAT	8		[20, 38, 41, 46, 110, 174, 175, 178]
	RGCN	4		[13, 31, 158, 180]
	HGNN	1		[103]
	RGAT	1		[30]
	DGCNN	1		[117]
	HGCN	1		[68]
	GCL	1		[144]
	BGNN	1		[10]
	GGRU	1		[157]
Convolutional Models	CNN	11	19	[17, 37, 48, 56, 73, 74, 79, 137, 138, 155, 190]
	TextCNN	6		[9, 64, 132, 141, 164, 176]
	TextRCNN	1		[30]
	QCNN	1		[60]
General Models	FCN	2	13	[81, 170]
	TCN	2		[16, 17]
	Auto Encoders	1		[71]
	Memory Neural Network	1		[23]
	GAN	1		[109]
	Feed Forward	1		[118]
	Representation Learning	1		[108]
	DRSN	1		[16]
	DCN	1		[76]
	Others	1		[4]
	DBN	1		[146]
Transformers	BERT	2	9	[93, 134]
	GraphCodeBERT	1		[153]
	CodeBERT	1		[111]
	HGT	1		[165]
	GPT-4	1		[98]
	GPT-3.5_turbo	1		[135]
	Code-T5	1		[169]
	Transformer Encoder	1		[171]
Attention Models	–	8	8	[22, 34, 49, 62, 80, 96, 140, 177]
Unique Total	–	–	**124**	–

Table 8. Distribution of Classic ML and Other Models in the Subject Studies

Category	Model Name	# Studies	Total	References
Classic ML Models	Random Forest	7	38	[19, 70, 94, 114, 122, 166, 184]
	Naive Bayes	5		[19, 70, 114, 122, 184]
	SVM	4		[19, 115, 122, 184]
	K-NN	4		[19, 95, 122, 184]
	Logistic Regression	3		[70, 114, 184]
	AdaBoost	3		[19, 33, 184]
	Decision Tree	2		[70, 122]
	Gradient Boosting	2		[19, 184]
	PCA	1		[162]
	Kernel Machine	1		[107]
	ADTree	1		[114]
	TAN	1		[70]
	Gradient Boosting Classifier	1		[33]
	SGDClassifier	1		[33]
	AdaBoostClassifier	1		[33]
	TrAdaBoost	1		[33]
Distance/Similarity Measures	–	3	3	[45, 58, 163]
Language Models	N-gram	1	1	[126]
Unique Total	–	–	**14**	–

95]. Second, software vulnerability detection often requires understanding domain-specific languages and contexts, which can vary widely between different applications and systems [33, 95].

Among the studies we reviewed, six studies utilized transfer learning for software vulnerability detection. Liu et al. [95] minimized distribution disparities between domains by improving cross-domain representations using a metric transfer learning framework). With this method, the model can still generalize well even in cases when the projects or vulnerability types in the test and training data are different. Du et al. [33] presented a system for detecting software vulnerabilities that makes use of the transfer learning algorithm TrAdaBoost. By using labeled bug reports from one project to predict issue categories in another where labeled data is insufficient, their method identifies bug types across several projects. Sendner et al. [125] customized transfer learning for smart contract software vulnerability detection. Their method, called *ESCORT*, uses a common feature extractor to understand the semantics of the bytecode, with different branches responding to different kinds of vulnerabilities. The transfer learning capability of ESCORT increases system flexibility by making it easier to include new vulnerability types with less data. Zhou et al. [182] presented a framework for adversarial multi-task learning that integrates common and task-specific components to maximize feature extraction while using adversarial transfer learning to reduce noise and interference between private and general features. Li et al. [77] explored the identification of cross-domain vulnerabilities using VulGDA, a system that combines graph embedding and deep-domain adaptation methods. To capture syntactic and semantic links and improve feature extraction through domain-invariant feature generation, VulGDA transforms samples of source code into graph representations. Zhang et al. [174] proposed CPVD, a cross-domain vulnerability detection method that utilizes labeled data from one source to accurately predict vulnerability labels. CPVD encodes code as property graphs and uses a graph attention network and convolution pooling network for feature extraction.

Answer to RQ3

(1) A total of 88.4% of the subject studies use DL models for vulnerability detection, whereas merely 7.2% of the subject studies use classic ML models.
(2) Recurrent and graph models are by far the most popular DL-based models in software vulnerability detection.
(3) BiLSTM is the most popular architecture in RNN-based models, and GCN is the most commonly used model in the graph-based category.
(5) Besides DL models, classic ML models are popular for software vulnerability detection. Random Forest is the most popular model, accounting for seven studies.

4.4 RQ4: What Is the Most Frequent Type of Vulnerability Covered in the Subject Studies?

Software vulnerability detection datasets support different vulnerability types. For example, NVD and SARD benchmarks together support 96 types of vulnerabilities. This RQ intends to summarize the most popular vulnerability types covered by subject studies and their frequency. Table 9 shows the statistics regarding the vulnerability types. The column *CWE-Type* indicates the type of CWE.[15] There are many categories on the CWE website for vulnerability categorization including *categorization by software development*, *categorization by hardware design*, and *categorization by research concepts*. The categorization shown in Table 9 is based on *categorization by research*

[15] https://cwe.mitre.org/

Table 9. Top Vulnerability Types Covered in the Subject Studies

Category	CWE-Type	Severity Score	# Studies	Total	References
Resource	CWE-119	–	29	121	[9, 11, 16, 20, 29, 30, 34, 36–38, 47, 49, 71, 75, 85, 87, 95, 98, 107, 110, 121, 131, 132, 140–142, 151, 159, 164]
	CWE-476	–	13		[11, 29, 30, 36, 47, 98, 110, 131, 132, 142, 151, 152, 159]
	CWE-399	–	13		[9, 16, 34, 37, 49, 75, 85, 98, 110, 131, 132, 141, 159]
	CWE-400	–	10		[9, 20, 30, 47, 132, 141–143, 159, 165]
	CWE-22	–	10		[9, 20, 30, 38, 41, 140–142, 151, 159]
	CWE-787	–	9		[9, 11, 20, 38, 98, 132, 141, 151, 165]
	CWE-125	–	9		[9, 11, 20, 98, 110, 132, 141, 151, 152]
	CWE-416		9		[11, 29, 30, 98, 110, 131, 132, 151, 159]
	CWE-122	–	7		[9, 11, 23, 121, 138, 141, 152]
	CWE-121	–	6		[11, 121, 138, 141, 152, 164]
	CWE-362	–	6		[98, 110, 131, 140, 142, 151]
Validation	CWE-20	–	13	37	[9, 20, 30, 38, 98, 110, 131, 132, 141, 142, 151, 159, 165]
	CWE-78		9		[9, 20, 41, 75, 83, 141, 142, 151, 165]
	CWE-841	–	8		[4, 8, 62, 125, 134, 168, 169, 171]
	CWE-200	–	7		[30, 38, 98, 131, 132, 140, 142]
Numeric	CWE-190		23	36	[4, 8, 9, 20, 29, 38, 58, 62, 98, 110, 120, 125, 131, 132, 140–143, 151, 152, 165, 168, 182]
	CWE-189		7		[30, 87, 98, 110, 131, 132, 190]
	CWE-191		6		[4, 125, 140, 142, 168, 182]
Unique Total	–	–	–	**48**	–

concepts, as this categorization is a perfect match for vulnerability types reported in the subject studies.

Table 9 indicates that the vulnerability category that receives the highest attendance is related to *Resource* vulnerabilities, mentioned in 121 studies. This category primarily involves managing a system's resources, which are created, utilized, and disposed of according to a pre-defined set of instructions. It is observable that CWE-119 [95, 107, 121] is the most frequent vulnerability type addressed by the subject studies. This vulnerability occurs when a software system attempts to access or write to a memory location outside the permitted boundary of the system's buffer. The second most frequent vulnerability type is Null Pointer Dereference (CWE-476), accounting for 13 subject studies. This vulnerability occurs when a program attempts to read or write to a memory location through a pointer that has not been properly initialized and points to NULL (no valid memory address).

Validation-related vulnerabilities is the second major family of vulnerability types, covered by 37 subject studies. In this type, the attackers exploit input and output data when they are malformed or not validated properly. As can be seen, CWE-20 [20, 142] is the most frequent type of vulnerability, accounting for 13 subject studies. CWE-20 refers to a situation where input validation is not done properly in software systems, making them vulnerable to attacks by malicious individuals who can exploit input data. This occurs when the input data is not verified to be safe or in line with the pre-defined specifications. CWE-78 is the second major vulnerability type, covered by 9 subject studies [11, 20, 38]. This category of security vulnerability pertains to OS command injection, in which an external attacker can construct an OS command by using input data from components that have not been adequately verified.

Vulnerabilities related to *Numeric* are the third most frequent type of vulnerabilities covered in the subject studies, accounting for 36 studies in total. Within this class, Integer Overflow (CWE-190) is the most frequently covered vulnerability type [20, 58, 142]. Integer overflow is a condition that occurs when an arithmetic operation attempts to create a numeric value that is outside the range that can be represented with a given number of bits. For example, an 8-bit unsigned integer can represent values from 0 to 255, whereas a 32-bit signed integer typically ranges from –2,147,483,648 to 2,147,483,647. When an arithmetic operation produces a value that exceeds these limits, an overflow occurs.

Answer to RQ4

(1) The most frequent type of vulnerabilities covered in the subject studies is *Resource*-related vulnerabilities. Improper Restriction of Operations within the Bounds of a Memory Buffer (CWE-119) is the most frequent type of vulnerability in this category, accounting for 29 subject studies.
(2) In the category of *Validation* vulnerabilities, Improper Input Validation (CWE-20) is the most frequently covered vulnerability type, accounting for 13 subject studies in total.
(3) Vulnerability types related to *Numeric* are the third most covered vulnerabilities within the subject studies, accounting for 36 studies in total. Within this category, Integer Overflow (CWE-190) is the vulnerability type that is covered by most subject studies.

4.5 RQ5: What Are the Most Frequently Used Tools for Software Vulnerability Detection?

In this section, we summarize the most commonly used tools for software vulnerability detection. Table 10 shows the distribution of the tools. We summarized the tools into three categories, including *Model Building Tools*, *Code Analysis/Compilation*, and *Data Tools*.

As can be seen in the table, Keras with TensorFlow backend[16] is the most commonly used library for building ML-based software vulnerability detection techniques, accounting for 42 studies, and PyTorch[17] comes as the second most commonly used library, with 42 studies in total. Scikit-learn[18] is the third most popular library for model building, accounting for 11 studies in total. Scikit-learn provides a user-friendly and consistent API, making it easy to implement and experiment with various ML algorithms. Scikit-learn includes a diverse set of classification algorithms such as Logistic Regression, SVM, Decision Trees, Random Forests, KN, and Naive Bayes. GenSim[19] is the fourth commonly used tool for building software vulnerability detection models. GenSim's ability to efficiently handle large datasets, combined with its powerful topic modeling and word embedding functionalities, makes it an indispensable tool for model building in natural language processing and text mining. DGL[20] is the fifth most commonly used model building tool, accounting for 6 studies. DGL is specifically designed for constructing and training GNNs, making it a go-to library for researchers and practitioners working on graph-related problems. It abstracts the complexity of implementing GNNs, providing easy-to-use APIs for building and applying various GNN models.

In the category of *Code Analysis/Compilation*, the most commonly used tool is Joern, accounting for 24 studies in total. Joern was first proposed by Yamaguchi et al. [160], and it converts source code into a graph representation, specifically AST, CFG, and PDG. The second most commonly used tool for code processing is Soot,[21] which provides various intermediate representations of Java bytecode.

In the category of *Data Tools*, NetworkX is the most commonly used data tool, accounting for five studies in total. NetworkX[22] uses native Python data structures (like dictionaries and lists) to represent graphs. This allows seamless integration with other Python libraries and makes it easy to manipulate and explore graph data. NLTK[23] provides robust tools for breaking down source code and text into tokens, which is essential for analyzing software vulnerability data.

[16] https://www.tensorflow.org/
[17] https://pytorch.org/
[18] https://scikit-learn.org
[19] https://radimrehurek.com/gensim/
[20] https://www.dgl.ai/
[21] https://soot-oss.github.io/soot/
[22] https://networkx.org/
[23] https://www.nltk.org/

Table 10. Most Commonly Used Tools for Software Vulnerability Detection

Category	Tool Name	# Studies	Total	References
Model Building Tools	Keras/TensorFlow	42	116	[10, 11, 16, 17, 23, 26, 29, 32, 37, 41, 60, 62–64, 68, 71, 74, 76, 83–85, 87, 95–97, 106, 107, 109, 114, 118, 125, 131, 139, 142, 144, 148, 149, 154, 164, 186, 189, 190]
	PyTorch	42		[3, 8, 9, 13, 20–22, 28, 30, 31, 38, 48, 49, 54, 57, 64, 68, 77, 82, 91, 120, 127, 129, 132, 138, 140, 141, 143, 143, 151, 152, 155, 170, 174, 175, 178–181, 183, 185, 188]
	Scikit-learn	11		[19, 33, 41, 54, 60, 62, 70, 87, 142, 174, 175]
	GenSim	9		[41, 54, 64, 87, 95, 140, 154, 174, 183]
	DGL	6		[7, 8, 129, 174, 175, 180]
	Theano	2		[26, 85]
	sent2vec	2		[155, 188]
	Transformers	2		[38, 48]
Code Analysis/Compilation	Joern	24	35	[9, 11, 22, 29, 30, 68, 71, 110, 129, 132, 137, 141–143, 147, 150, 152, 155, 170, 174, 175, 183, 188, 189]
	Soot	3		[79, 80, 142]
	Clang	2		[22, 48]
	tree-sitter	2		[152, 153]
	CodeSensor	2		[94, 95]
	ANTLR	2		[135, 142]
Data Tools	NetworkX	5	9	[7, 9, 41, 141, 155]
	NLTK	4		[54, 56, 73, 114]
Unique Total	–	–	**96**	–

Answer to RQ5

(1) Keras with TensorFlow backend is the most commonly used library for building ML-based software vulnerability detection techniques, followed closely by PyTorch, with 42 studies for each. Scikit-learn, used in 11 studies, is known for its user-friendly API, diverse classification algorithms, robust preprocessing tools, and strong model evaluation capabilities, making it a popular choice for building classification models.
(2) In the category of *Code Analysis/Compilation*, Joern is the most commonly used tool because of its effective graph-based code representations. Soot, the second most used tool, provides detailed analysis through various intermediate representations of Java bytecode.
(3) In the category of *Data Tools*, NetworkX and NLTK are the most widely used tools, accounting for 5 and 4 studies, respectively.

4.6 RQ6: What Are Possible Challenges and Open Directions in Software Vulnerability Detection?

4.6.1 Challenges.

Challenge 1: Heterogeneous Data Sources. The biggest challenge in vulnerability detection through learning is the inadequate modeling of the comprehensive semantics of complex vulnerabilities by current models [26, 27, 126]. Existing ML models often fail to capture the complex patterns of software vulnerabilities because they treat source code like natural language. Unlike natural language, source code contains structural and logical information requiring AST, dataflow, and control flow analysis. To address this, the detection pipeline must use rich representation techniques like control flow and dataflow graphs and proper embeddings to convert these representations into a numerical format for graph-based neural networks.

Challenge 2: Detection Granularity. The effectiveness of DL models in identifying vulnerabilities depends on input granularity. Current models use coarse inputs like methods and files. To achieve finer granularity, program slicing can select crucial statements for detection, but it must be done effectively to reduce noise. Existing tools focus on library/API calls and operations, but these alone are insufficient. A promising approach is using code changes from GitHub, focusing on added and deleted lines, which often have the highest impact on vulnerability detection.

Challenge 3: Lack of Training Data. A significant weakness of DL models, particularly in software vulnerability detection, is their insatiable need for data [24, 111]. In domains like image classification, abundant labeled data, and pre-trained models enable effective DL training. However, in software vulnerability detection, data scarcity is a major issue due to the difficulty of labeling ground truth information. Platforms like Stack Overflow, GitHub, and issue-tracking systems provide extensive records, but labeling is often manual and challenging. Automatic labeling is a potential solution but tends to generate many false positives. Some researchers use unsupervised classification, but this method also has limited precision.

Answer to RQ6: Challenges

(1) Most of the current models cannot capture the comprehensive semantics of complex vulnerabilities, as most of them fail to consider the structural and logical information present in source code snippets.
(2) Most existing ML models process source code in a linear sequential manner, which limits their ability to identify intricate vulnerability patterns.
(3) DL models require a significant amount of labeled data for effective training. However, in software vulnerability detection, labeled data are scarce due to the challenging task of manual labeling. Automatic labeling approaches often generate false positives, and unsupervised classification suffers from limited precision.

4.6.2 Open Directions. ***Multi-Modal Learning***. Performing a simple vulnerability detection with source code snippets is not sufficient to have accurate and effective models. Various artifacts are needed to feed into ML models to increase vulnerability detection performance. For example, feeding code comments will increase classification performance remarkably. Some subject studies that use commits [19] argue that feeding source code is not enough and commit characteristics as metadata are required for software vulnerability detection.

Just-in-Time Vulnerability Detection. One possible direction for software vulnerability detection is using the just-in-time approaches. This approach focuses on detecting vulnerabilities as they occur or are introduced, hence offering real-time protection [56, 114]. This method allows for faster reaction and mitigation of vulnerabilities before they are exploited.

Leveraging Foundation Models (LLMs) for Vulnerability Detection. Recently, LLMs have been used in a wide variety of software engineering tasks including automatic program repair [59], test case generation, and root cause analysis of incidents in cloud environments. However, the application of LLMs for software vulnerability detection has not been yet discovered comprehensively as it should be. In our survey, we identified some subject studies that utilize LLMs for software vulnerability detection [32, 98, 135, 169]. However, their frequency is still negligible compared to the widespread usage of typical DL models.

Answer to RQ6: Opportunities

(1) Leveraging LLMs for software vulnerability detection is a promising opportunity due to their advanced understanding of both natural language and code. LLMs can help with code analysis and recognize emerging vulnerability patterns.
(2) Multi-modal learning offers a significant opportunity to enhance software vulnerability detection by integrating diverse data sources, such as source code, natural language comments, metadata, and runtime behaviors.

5 Threats to Validity

In this section, we discuss threats to the validity of each RQ. We discuss various threats to the RQs that we address in this study.

RQ1: Trend of Studies. The selection of studies might be biased if certain types of studies are more likely to be indexed or retrieved by our web crawler. To address selection bias, we defined diverse key terms to extract the most relevant research papers related to software vulnerability detection. The target papers should use ML-based software vulnerability detection techniques. To increase the accuracy of data selection, we refined the initial search results in three steps to ensure that the most relevant studies were selected for taxonomy creation and refinement. These steps have been performed by multiple authors simultaneously. The choice of digital libraries could impact construct validity if they do not equally represent all relevant studies. To mitigate this threat, we selected the most widely used digital libraries: ACM Digital Library, ScienceDirect, IEEE Xplore, and Google Scholar. These libraries are representative of the software vulnerability detection field because they contain a sufficient number of records that match our key terms for data extraction. One of the major threats to the external validity of the first RQ is that the trends we observed from January 2011 to June 2024 may not apply to future research beyond this period. As technologies evolve rapidly, new techniques and tools for software vulnerability detection may emerge. However, we believe that our findings accurately represent the current state-of-the-art technology for software vulnerability detection at the time of this study.

RQ2: Characteristics of Software Vulnerability Detection Datasets. Datasets might focus on specific types of software or languages that threaten the generalizability of our findings. To overcome this limitation, we focused on software vulnerability detection in three major language domains, including software vulnerability in Java, C/C++, and smart contracts. Java is prevalent in enterprise and web applications, C/C++ is fundamental in system and performance-critical programming, and smart contracts are crucial in blockchain technology. This diverse selection reduces selection bias, provides a holistic view of vulnerabilities, and ensures that the findings are more broadly applicable and relevant to real-world software development contexts. Although our findings are based on datasets from studies published between January 2011 and June 2024, the identified characteristics are expected to apply to future datasets due to ongoing advancements in software vulnerability detection techniques. We provide detailed criteria and procedures for selecting and analyzing datasets, enabling other researchers to replicate and validate our findings, thus enhancing the generalizability and reliability of our conclusions.

RQ3: Distribution of ML and DL Models in Software Vulnerability Detection. There are multiple threats to this RQ. First, ML models evolve quickly, and models that are effective today might become obsolete or be replaced by more advanced ones soon. To overcome this threat, we expanded our study selection bias to cover the last 2 years—that is, 2023 and 2024 to cover the most state-of-the-art ML technology for software vulnerability detection. This results in identifying three promising studies that use foundation models for software vulnerability detection.

RQ4: Frequent Software Vulnerability. To ensure construct validity in this RQ, it is crucial to provide clear and precise definitions of each type of vulnerability. We first identify reputable sources like OWASP and MITRE's CWE. OWASP provides a widely recognized list of common security vulnerabilities, particularly in web applications. CWE offers a comprehensive list of software weaknesses, providing detailed descriptions and classifications. We then reviewed the subject studies and identified the types of vulnerabilities that are mentioned frequently. Often these vulnerabilities can be identified by CWE IDs that are explicitly mentioned in the research papers.

RQ5: Tools for Software Vulnerability Detection. The threat to this question is that there may be biases in the selection of tools for study, influenced by an important factor such as popularity (like TensorFlow and PyTorch). This can skew the findings toward more well-known tools, neglecting equally effective but less publicized options. To overcome this threat, we classified the tools into three broad categories. For each category, we extracted the most popular and the least popular tools including a balanced mix of tools to avoid over-representation of any particular subset.

RQ6: Challenges and Open Directions. To ensure the construct validity of this RQ, we thoroughly analyzed two key sections of each study. First, we examined the context section of the abstract to gain a general understanding of the problem being addressed. Next, we analyzed the introduction section to extract relevant text that further elaborates on the problem. By combining this information, we generalized the problem and created a concise taxonomy for classification.

6 Conclusion

In this study, we conducted a systematic survey to investigate various characteristics of ML-based software vulnerability detection studies using six RQs. We extracted initial studies from four widely-used online digital libraries—ACM Digital Library, IEEE Xplore, ScienceDirect, and Google Scholar—using a custom web scraper. After manually filtering out irrelevant studies unrelated to software vulnerability detection, we created taxonomies and addressed the RQs.

Our findings indicated a notable increase in the use of ML techniques to detect software vulnerabilities in recent years. We found that prominent conference venues include ICSE, ISSRE, MSR, and FSE, whereas the leading journal venues are IST, C&S, and JSS. Additionally, we found that 39.1% of the subject studies use hybrid as the sources of data, whereas 37.6% of the subject studies use benchmark data for software vulnerability detection. Among the data types analyzed, code-based data is the most prevalent, with source code being the most common sub-type. Graph-based and token-based input representations are the most popular techniques, utilized in 57.2% and 24.6% of the studies, respectively. For input embedding, graph embeddings and token vector embeddings are the most frequently employed methods, appearing in 32.6% and 29.7% of studies. Furthermore, 88.4% of the examined studies use DL models, with RNNs and GNNs being the most popular, whereas only 7.2% use traditional ML models. The most frequently addressed vulnerability types are CWE-119, CWE-20, and CWE-190. In terms of tools for software vulnerability detection, Keras and PyTorch are the most widely used tools. Joern is the leading tool for code analysis and representation. Finally, we summarized the challenges and future directions in the context of software vulnerability detection, providing valuable insight for researchers and practitioners in the field. This comprehensive survey aimed to bridge the existing gap and provide a clearer understanding of the current landscape and future opportunities in the detection of software vulnerabilities using ML techniques.

References

[1] Faranak Abri, Sima Siami-Namini, Mahdi Adl Khanghah, Fahimeh Mirza Soltani, and Akbar Siami Namin. 2019. Can machine/deep learning classifiers detect zero-day malware with high accuracy? In *Proceedings of the 2019 IEEE International Conference on Big Data (Big Data'19)*. IEEE, 3252–3259.

[2] Sasan H. Alizadeh, Alireza Hediehloo, and Nima Shiri Harzevili. 2021. Multi independent latent component extension of naive Bayes classifier. *Knowledge-Based Systems* 213 (2021), 106646.

[3] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. In *Proceedings of the 35th Conference on Neural Information Processing Systems (NeurIPS'21)*. 27865–27876.

[4] Nami Ashizawa, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. 2021. Eth2Vec: Learning contract-wide code representations for vulnerability detection on Ethereum smart contracts. In *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure (BSCI'21)*. 47–59.

[5] Leyla Bilge and Tudor Dumitraş. 2012. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCCS'12)*. 833–844.

[6] Christopher M. Bishop and Nasser M. Nasrabadi. 2006. *Pattern Recognition and Machine Learning*. New York: Springer, 4, 4 (2006).

[7] Jie Cai, Bin Li, Jiale Zhang, Xiaobing Sun, and Bing Chen. 2023. Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. *Journal of Systems and Software* 195 (2023), 111550.

[8] Jie Cai, Bin Li, Tao Zhang, Jiale Zhang, and Xiaobing Sun. 2024. Fine-grained smart contract vulnerability detection by heterogeneous code feature learning and automated dataset construction. *Journal of Systems and Software* 209 (2024), 111919.

[9] Wenjing Cai, Junlin Chen, Jiaping Yu, and Lipeng Gao. 2023. A software vulnerability detection method based on deep learning with complex network analysis and subgraph partition. *Information and Software Technology* 164 (2023), 107328.

[10] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. BGNN4VD: Constructing bidirectional graph neural-network for vulnerability detection. *Information and Software Technology* 136 (2021), 106576.

[11] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. 1456–1468.

[12] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* 48 (2022), 3280–3296.

[13] Da Chen, Lin Feng, Yuqi Fan, Siyuan Shang, and Zhenchun Wei. 2023. Smart contract vulnerability detection based on semantic graph and residual graph convolutional networks with edge attention. *Journal of Systems and Software* 202 (2023), 111705.

[14] Haipeng Chen, Jing Liu, Rui Liu, Noseong Park, and V. S. Subrahmanian. 2019. VEST: A system for vulnerability exploit scoring & timing. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*. 6503–6505.

[15] Jinfu Chen, Patrick Kwaku Kudjo, Solomon Mensah, Selasie Aformaley Brown, and George Akorfu. 2020. An automatic software vulnerability classification framework using term frequency-inverse gravity moment and feature selection. *Journal of Systems and Software* 167 (2020), 110616.

[16] Jinfu Chen, Wei Lin, Saihua Cai, Yemin Yin, Haibo Chen, and Dave Towey. 2023. BiTCN_DRSN: An effective software vulnerability detection model based on an improved temporal convolutional network. *Journal of Systems and Software* 204 (2023), 111772.

[17] Jinfu Chen, Weijia Wang, Bo Liu, Saihua Cai, Dave Towey, and Shengran Wang. 2024. Hybrid semantics-based vulnerability detection incorporating a temporal convolutional network and self-attention mechanism. *Information and Software Technology* 171 (2024), 107453.

[18] Stanley F. Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language* 13, 4 (1999), 359–394.

[19] Yang Chen, Andrew E. Santosa, Ang Ming Yi, Abhishek Sharma, Asankhaya Sharma, and David Lo. 2020. A machine learning approach for vulnerability curation. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR'20)*. 32–42.

[20] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology* 30, 3 (2021), 1–33.

[21] Xiao Cheng, Haoyu Wang, Jiayi Hua, Miao Zhang, Guoai Xu, Li Yi, and Yulei Sui. 2019. Static detection of control-flow-related vulnerabilities using graph embedding. In *Proceedings of the 2019 24th International Conference on Engineering and Complex Computer Systems (ICECCS'19)*. IEEE, 41–50.

[22] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22)*. 519–531.

[23] Min-Je Choi, Sehun Jeong, Hakjoo Oh, and Jaegul Choo. 2017. End-to-end prediction of buffer overruns from raw source code via neural memory networks. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*. 1546–1553.

[24] Roland Croft, M. Ali Babar, and M. Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*. IEEE, 121–133.
[25] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. 2008. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology* 17, 2 (2008), Article 8, 37 pages.
[26] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2018. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering* 47, 1 (2018), 67–85.
[27] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2018. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering* 47, 1 (2018), 67–85.
[28] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *Proceedings of the 2020 International Conference on Learning Representations (ICLR'20)*.
[29] Yangruibo Ding, Sahil Suneja, Yunhui Zheng, Jim Laredo, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. 2022. VELVET: A noVel Ensemble Learning approach to automatically locate VulnErable sTatements. In *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'22)*. 959–970.
[30] Yukun Dong, Yeer Tang, Xiaotong Cheng, and Yufei Yang. 2023. DeKeDVer: A deep learning-based multi-type software vulnerability classification framework using vulnerability description and source code. *Information and Software Technology* 163 (2023), 107290.
[31] Yukun Dong, Yeer Tang, Xiaotong Cheng, Yufei Yang, and Shuqi Wang. 2023. SedSVD: Statement-level software vulnerability detection based on relational graph convolutional network with subgraph embedding. *Information and Software Technology* 158 (2023), 107168.
[32] Xiaozhi Du, Shiming Zhang, Yanrong Zhou, and Hongyuan Du. 2024. A vulnerability severity prediction method based on bimodal data and multi-task learning. *Journal of Systems and Software* 213 (2024), 112039.
[33] Xiaoting Du, Zenghui Zhou, Beibei Yin, and Guanping Xiao. 2020. Cross-project bug type prediction based on transfer learning. *Software Quality Journal* 28, 1 (2020), 39–57.
[34] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. 2019. VulSniper: Focus your attention to shoot fine-grained vulnerabilities. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*. 4665–4671.
[35] Facebook. 2013. Infer. Retrieved October 12, 2024 from https://fbinfer.com/
[36] Yuanhai Fan, Chuanhao Wan, Cai Fu, Lansheng Han, and Hao Xu. 2023. VDoTR: Vulnerability detection based on tensor representation of comprehensive code graphs. *Computers & Security* 130 (2023), 103247.
[37] Katarzyna Filus, Miltiadis Siavvas, Joanna Domańska, and Erol Gelenbe. 2020. The random neural network as a bonding model for software vulnerability prediction. In *Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. Lecture Notes in Computer Science, Vol. 12527. Springer, 102–116.
[38] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR'22)*.
[39] Cuifeng Gao, Wenzhang Yang, Jiaming Ye, Yinxing Xue, and Jun Sun. 2024. sGuard+: Machine learning guided rule-based automated vulnerability repair on smart contracts. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–55.
[40] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys* 50, 4 (2017), 1–36.
[41] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2021. Neural software vulnerability analysis using rich intermediate graph representations of programs. *Information Sciences* 553 (2021), 189–207.
[42] Patrice Godefroid. 2007. Random testing for security: Blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd International Conference on Random Testing, Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. 1.
[43] Xi Gong, Zhenchang Xing, Xiaohong Li, Zhiyong Feng, and Zhuobing Han. 2019. Joint prediction of multiple vulnerability characteristics through multi-task learning. In *Proceedings of the 2019 24th International Conference on Engineering and Complex Computer Systems (ICECCS'19)*. IEEE, 31–40.
[44] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative adversarial networks. *Communications of the ACM* 63, 11 (2020), 139–144.
[45] Yeming Gu, Hui Shu, and Fei Kang. 2023. BinAIV: Semantic-enhanced vulnerability detection for Linux x86 binaries. *Computers & Security* 135 (2023), 103508.
[46] Longtao Guo, Huakun Huang, Sihun Xue, Peiliang Wang, and Lingjun Zhao. 2023. Reentrancy vulnerability detection based on graph convolutional networks and expert patterns. In *Proceedings of the 2023 IEEE 16th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip (MCSoC'23)*. 312–316.

[47] Wenbo Guo, Yong Fang, Cheng Huang, Haoran Ou, Chun Lin, and Yongyan Guo. 2022. HyVulDect: A hybrid semantic vulnerability mining system based on graph neural network. *Computers & Security* 121 (2022), 102823.

[48] Hazim Hanif and Sergio Maffeis. 2022. Vulberta: Simplified source code pre-training for vulnerability detection. In *Proceedings of the 2022 International Joint Conference on Neural Networks (IJCNN'22)*. IEEE, 1–8.

[49] M. Hariharan, C. Sathish Kumar, Anshul Tanwar, Krishna Sundaresan, Prasanna Ganesan, Sriram Ravi, and R. Karthik. 2022. Proximal instance aggregator networks for explainable security vulnerability detection. *Future Generation Computer Systems* 134 (2022), 303–318.

[50] Nima Shiri Harzevili and Sasan H. Alizadeh. 2018. Mixture of latent multinomial naive Bayes classifier. *Applied Soft Computing* 69 (2018), 516–527.

[51] Nima Shiri Harzevili and Sasan H. Alizadeh. 2021. Analysis and modeling conditional mutual dependency of metrics in software defect prediction using latent variables. *Neurocomputing* 460 (2021), 309–330.

[52] Nima Shiri Harzevili, Jiho Shin, Junjie Wang, Song Wang, and Nachiappan Nagappan. 2023. Automatic static vulnerability detection for machine learning libraries: Are we there yet? In *Proceedings of the 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE'23)*. IEEE, 795–806.

[53] Nima Shiri Harzevili, Jiho Shin, Junjie Wang, Song Wang, and Nachiappan Nagappan. 2023. Characterizing and understanding software security vulnerabilities in machine learning libraries. In *Proceedings of the 20th International Conference on Mining Software Repositories (MSR'23)*. IEEE, 27–38.

[54] David Hin, Andrey Kan, Huaming Chen, and M. Ali Babar. 2022. LineVD: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR'22)*. 596–607.

[55] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. 2006. A fast learning algorithm for deep belief nets. *Neural Computation* 18, 7 (2006), 1527–1554.

[56] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR'19)*. IEEE, 34–45.

[57] Huakun Huang, Longtao Guo, Lingjun Zhao, Haoda Wang, Chenkai Xu, and Shan Jiang. 2024. Effective combining source code and opcode for accurate vulnerability detection of smart contracts in edge AI systems. *Applied Soft Computing* 158 (2024), 111556.

[58] Jianjun Huang, Songming Han, Wei You, Wenchang Shi, Bin Liang, Jingzheng Wu, and Yanjun Wu. 2021. Hunting vulnerable smart contracts via graph embedding based bytecode matching. *IEEE Transactions on Information Forensics and Security* 16 (2021), 2144–2156.

[59] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE'23)*. IEEE, 1162–1174.

[60] Shumaila Hussain, Muhammad Nadeem, Junaid Baber, Mohammed Hamdi, Adel Rajab, Mana Saleh Al Reshan, and Asadullah Shaikh. 2024. Vulnerability detection in Java source code using a quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction. *Scientific Reports* 14, 1 (2024), 7406.

[61] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2022. The secret life of software vulnerabilities: A large-scale empirical study. *IEEE Transactions on Software Engineering* 49, 1 (2022), 44–63.

[62] Vikas Kumar Jain and Meenakshi Tripathi. 2023. Multi-objective approach for detecting vulnerabilities in Ethereum smart contracts. In *Proceedings of the 2023 International Conference on Emerging Trends in Networks and Computer Communications (ETNCC'23)*. IEEE, 1–6.

[63] Sanghoon Jeon and Huy Kang Kim. 2021. AutoVAS: An automated vulnerability analysis system with a deep learning approach. *Computers & Security* 106 (2021), 102308.

[64] Wanqing Jie, Qi Chen, Jiaqi Wang, Arthur Sandor Voundi Koe, Jin Li, Pengfei Huang, Yaqi Wu, and Yin Wang. 2023. A novel extended multimodal AI framework towards vulnerability detection in smart contracts. *Information Sciences* 636 (2023), 118907.

[65] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. EBSE Technical Report, Version 2.3. EBSE.

[66] Saad Khan and Simon Parkinson. 2018. Review into state of the art of vulnerability assessment using artificial intelligence. In *Guide to Vulnerability Analysis for Computer Networks and Systems*. Springer, 3–32.

[67] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. 2019. RVFuzzer: Finding input validation bugs in robotic vehicles through control-guided testing. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. 425–442.

[68] Lingdi Kong, Senlin Luo, Limin Pan, Zhouting Wu, and Xinshuai Li. 2024. A multi-type vulnerability detection framework with parallel perspective fusion and hierarchical feature enhancement. *Computers & Security* 140 (2024), 103787.

[69] Kyriakos Kritikos, Kostas Magoutis, Manos Papoutsakis, and Sotiris Ioannidis. 2019. A survey on vulnerability assessment tools and databases for cloud-based web applications. *Array* 3 (2019), 100011.

[70] Jorrit Kronjee, Arjen Hommersom, and Harald Vranken. 2018. Discovering software vulnerabilities using data-flow analysis and machine learning. In *Proceedings of the 13th International Conference on Availability, Reliability, and Security (ARES'18)*. 1–10.

[71] Tue Le, Tuan Nguyen, Trung Le, Dinh Phung, Paul Montague, Olivier De Vel, and Lizhen Qu. 2018. Maximal divergence sequential autoencoder for binary software vulnerability detection. In *Proceedings of the 2018 International Conference on Learning Representations (ICLR'18)*.

[72] Triet H. M. Le, Huaming Chen, and M. Ali Babar. 2022. A survey on data-driven software vulnerability assessment and prioritization. *ACM Computing Surveys* 55, 5 (2022), 1–39.

[73] 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 *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*. 717–729.

[74] Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu. 2017. Software defect prediction via convolutional neural network. In *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability, and Security (QRS'17)*. IEEE, 318–328.

[75] Litao Li, Steven H. H. Ding, Yuan Tian, Benjamin C. M. Fung, Philippe Charland, Weihan Ou, Leo Song, and Congwei Chen. 2023. VulANalyzeR: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution. *ACM Transactions on Privacy and Security* 26, 3 (2023), 1–25.

[76] Lina Li, Yang Liu, Guodong Sun, and Nianfeng Li. 2024. Smart contract vulnerability detection based on automated feature extraction and feature interaction. *IEEE Transactions on Knowledge and Data Engineering* 36, 9 (2024), 4916–4929.

[77] Xin Li, Yang Xin, Hongliang Zhu, Yixian Yang, and Yuling Chen. 2023. Cross-domain vulnerability detection using graph embedding and domain adaptation. *Computers & Security* 125 (2023), 103017.

[78] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. 292–303.

[79] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), Article 162, 30 pages. https://doi.org/10.1145/3360588

[80] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), Article 162, 30 pages.

[81] Yi Li, Aashish Yadavally, Jiaxing Zhang, Shaohua Wang, and Tien N. Nguyen. 2023. Commit-level, neural vulnerability detection and assessment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'23)*. 1024–1036.

[82] Zhaoxuan Li, Siqi Lu, Rui Zhang, Ziming Zhao, Rujin Liang, Rui Xue, Wenhao Li, Fan Zhang, and Sheng Gao. 2023. VulHunter: Hunting vulnerable smart contracts at EVM bytecode-level via multiple instance learning. *IEEE Transactions on Software Engineering* 49, 11 (2023), 4886–4916.

[83] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2022. VulDeeLocator: A deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing* 19 (2022), 2821–2837.

[84] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19 (2022), 2244–2258.

[85] 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 *Proceedings of the 2018 Network and Distributed Systems Security Symposium (NDSS'18)*. 1–15.

[86] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. 2020. Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE* 108, 10 (2020), 1825–1848.

[87] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Olivier De Vel, Paul Montague, and Yang Xiang. 2019. Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2019), 2469–2485.

[88] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yang Xiang, Olivier De Vel, and Paul Montague. 2018. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Transactions on Industrial Informatics* 14, 7 (2018), 3289–3297.

[89] Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. 2021. Traceability transformed: Generating more accurate links with pre-trained BERT models. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE'21)*. IEEE, 324–335.

[90] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. On the reproducibility and replicability of deep learning in software engineering. *ACM Transactions on Software Engineering and Methodology* 31, 1 (2021), 1–46.

[91] Haiyang Liu, Yuqi Fan, Lin Feng, and Zhenchun Wei. 2023. Vulnerable smart contract function locating based on multi-relational nested graph convolutional network. *Journal of Systems and Software* 204 (2023), 111775.

[92] Huijiang Liu, Shuirou Jiang, Xuexin Qi, Yang Qu, Hui Li, Tingting Li, Cheng Guo, and Shikai Guo. 2024. Detect software vulnerabilities with weight biases via graph neural networks. *Expert Systems with Applications* 238 (2024), 121764.

[93] Jingqiang Liu, Xiaoxi Zhu, Chaoge Liu, Xiang Cui, and Qixu Liu. 2022. CPGBERT: An effective model for defect detection by learning program semantics via code property graph. In *Proceedings of the 2022 IEEE International Conference on Trust, Security, and Privacy in Computing and Communications (TrustCom'22)*. IEEE, 274–282.

[94] Shigang Liu, Guanjun Lin, Qing-Long Han, Sheng Wen, Jun Zhang, and Yang Xiang. 2019. DeepBalance: Deep-learning and fuzzy oversampling for vulnerability detection. *IEEE Transactions on Fuzzy Systems* 28, 7 (2019), 1329–1343.

[95] Shigang Liu, Guanjun Lin, Lizhen Qu, Jun Zhang, Olivier De Vel, Paul Montague, and Yang Xiang. 2020. CD-VulD: Cross-domain vulnerability discovery based on deep domain adaptation. *IEEE Transactions on Dependable and Secure Computing* 19, 1 (2020), 438–451.

[96] Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. 2021. Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI'21)*.

[97] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2023. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering* 35, 2 (2023), 1296–1310.

[98] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software* 212 (2024), 112031.

[99] Abu Sayed Mahfuz. 2016. *Software Quality Assurance: Integrating Testing, Security, and Audit*. CRC Press.

[100] Yi Mao, Yun Li, Jiatai Sun, and Yixin Chen. 2020. Explainable software vulnerability detection based on attention-based bidirectional recurrent neural networks. In *Proceedings of the 2020 IEEE International Conference on Big Data (Big Data'20)*. IEEE, 4651–4656.

[101] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. 2013. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'13)*. IEEE, 65–74.

[102] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices* 42, 6 (2007), 89–100.

[103] Hoang H. Nguyen, Nhat-Minh Nguyen, Hong-Phuc Doan, Zahra Ahmadi, Thanh-Nam Doan, and Lingxiao Jiang. 2022. MANDO-GURU: Vulnerability detection for smart contract source code by heterogeneous graph embeddings. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'22)*. 1736–1740.

[104] Hoang H. Nguyen, Nhat-Minh Nguyen, Chunyao Xie, Zahra Ahmadi, Daniel Kudendo, Thanh-Nam Doan, and Lingxiao Jiang. 2022. MANDO: Multi-level heterogeneous graph embeddings for fine-grained detection of smart contract vulnerabilities. In *Proceedings of the 2022 IEEE 9th International Conference on Data Science and Advanced Analytics (DSAA'22)*. IEEE, 1–10.

[105] Hoang H. Nguyen, Nhat-Minh Nguyen, Chunyao Xie, Zahra Ahmadi, Daniel Kudendo, Thanh-Nam Doan, and Lingxiao Jiang. 2023. MANDO-HGT: Heterogeneous graph transformers for smart contract vulnerability detection. In *Proceedings of the 20th International Conference on Mining Software Repositories (MSR'23)*. IEEE, 334–346.

[106] Son Nguyen, Thu-Trang Nguyen, Thanh Trong Vu, Thanh-Dat Do, Kien-Tuan Ngo, and Hieu Dinh Vo. 2024. Code-centric learning-based just-in-time vulnerability detection. *Journal of Systems and Software* 214 (2024), 112014.

[107] Tuan Nguyen, Trung Le, Khanh Nguyen, Olivier de Vel, Paul Montague, John Grundy, and Dinh Phung. 2020. Deep cost-sensitive kernel machine for binary software vulnerability detection. In *Advances in Knowledge Discovery and Data Mining*. Lecture Notes in Computer Science, Vol. 12085. Springer, 164–177.

[108] Thu-Trang Nguyen and Hieu Dinh Vo. 2024. Context-based statement-level vulnerability localization. *Information and Software Technology* 169 (2024), 107406.

[109] Van Nguyen, Trung Le, Chakkrit Tantithamthavorn, John Grundy, and Dinh Phung. 2024. Deep domain adaptation with max-margin principle for cross-project imbalanced software vulnerability detection. *ACM Transactions on Software Engineering and Methodology* 33, 6 (2024), Article 162, 34 pages.

[110] Chao Ni, Xinrong Guo, Yan Zhu, Xiaodan Xu, and Xiaohu Yang. 2023. Function-level vulnerability detection through fusing multi-modal knowledge. In *Proceedings of the 2023 IEEE/ACM International Conference on Automated Software Engineering (ASE'23)*. IEEE, 1911–1918.

[111] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. 2022. The best of both worlds: Integrating semantic features with expert features for defect prediction and localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'22)*. 672–683.

[112] Yu Nong, Rainy Sharma, Abdelwahab Hamou-Lhadj, Xiapu Luo, and Haipeng Cai. 2022. Open science in software engineering: A study on deep learning-based vulnerability detection. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1983–2005.

[113] Shengyi Pan, Lingfeng Bao, Xin Xia, David Lo, and Shanping Li. 2023. Fine-grained commit-level vulnerability type prediction by CWE tree structure. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*. IEEE, 957–969.

[114] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software* 150 (2019), 22–36.

[115] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 426–437.

[116] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology* 64 (2015), 1–18.

[117] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. 2017. Convolutional neural networks over control flow graphs for software defect prediction. In *Proceedings of the 2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI'17)*. IEEE, 45–52.

[118] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), Article 147, 25 pages.

[119] Ali Raza and Waseem Ahmed. 2022. Threat and vulnerability management life cycle in operating systems: A systematic review. *Journal of Multidisciplinary Engineering Science and Technology* 9, 1 (2022), 15010–15013.

[120] Xiaojun Ren, Yongtang Wu, Jiaqing Li, Dongmin Hao, and Muhammad Alam. 2023. Smart contract vulnerability detection based on a semantic code structure and a self-designed neural network. *Computers and Electrical Engineering* 109 (2023), 108766.

[121] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA'18)*. IEEE, 757–762.

[122] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering* 40, 10 (2014), 993–1006.

[123] Hinrich Schütze, Christopher D. Manning, and Prabhakar Raghavan. 2008. *Introduction to Information Retrieval*. Vol. 39. Cambridge University Press, Cambridge.

[124] Abubakar Omari Abdallah Semasaba, Wei Zheng, Xiaoxue Wu, and Samuel Akwasi Agyemang. 2020. Literature survey of deep learning-based vulnerability analysis on source code. *IET Software* 14, 6 (2020), 654–664.

[125] Christoph Sendner, Huili Chen, Hossein Fereidooni, Lukas Petzi, Jan König, Jasper Stang, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2023. Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning. In *Proceedings of the 2023 Network and Distributed Security Symposium (NDSS'23)*.

[126] Thomas Shippey, David Bowes, and Tracy Hall. 2019. Automatically identifying code features for software defect prediction: Using AST n-grams. *Information and Software Technology* 106 (2019), 142–160.

[127] Zihua Song, Junfeng Wang, Kaiyuan Yang, and Jigang Wang. 2023. HGIVul: Detecting inter-procedural vulnerabilities based on hypergraph convolution. *Information and Software Technology* 160 (2023), 107219.

[128] Miroslaw Staron, Mirosław Ochodek, Wilhelm Meding, and Ola Söder. 2020. Using machine learning to identify code fragments for manual review. In *Proceedings of the 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'20)*. IEEE, 513–516.

[129] Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th International Conference on Software Engineering (ICSE'24)*. 1–13.

[130] Octavian Suciu, Connor Nelson, Zhuoer Lyu, Tiffany Bao, and Tudor Dumitraş. 2022. Expected exploitability: Predicting the development of functional vulnerability exploits. In *Proceedings of the 31st USENIX Security Symposium (Security'22)*. 377–394.

[131] Hao Sun, Lei Cui, Lun Li, Zhenquan Ding, Zhiyu Hao, Jiancong Cui, and Peng Liu. 2021. VDSimilar: Vulnerability detection based on code similarity of vulnerabilities and patches. *Computers & Security* 110 (2021), 102417.

[132] Hao Sun, Lei Cui, Lun Li, Zhenquan Ding, Siyuan Li, Zhiyu Hao, and Hongsong Zhu. 2024. VDTriplet: Vulnerability detection with graph semantics using triplet model. *Computers & Security* 139 (2024), 103732.

[133] Nan Sun, Jun Zhang, Paul Rimba, Shang Gao, Leo Yu Zhang, and Yang Xiang. 2018. Data-driven cybersecurity incident prediction: A survey. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1744–1772.

[134] Xiaobing Sun, Liangqiong Tu, Jiale Zhang, Jie Cai, Bin Li, and Yu Wang. 2023. ASSBert: Active and semi-supervised bert for smart contract vulnerability detection. *Journal of Information Security and Applications* 73 (2023), 103423.

[135] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. In *Proceedings of the 46th International Conference on Software Engineering (ICSE'24)*. 1–13.

[136] Wei Tang, Mingwei Tang, Minchao Ban, Ziguo Zhao, and Mingjun Feng. 2023. CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. *Journal of Systems and Software* 199 (2023), 111623.

[137] Zhiquan Tang, Qiao Hu, Yupeng Hu, Wenxin Kuang, and Jiongyi Chen. 2022. SeVulDet: A semantics-enhanced learnable vulnerability detector. In *Proceedings of the 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'22)*. IEEE, 150–162.

[138] Wenxin Tao, Xiaohong Su, Jiayuan Wan, Hongwei Wei, and Weining Zheng. 2023. Vulnerability detection through cross-modal feature enhancement and fusion. *Computers & Security* 132 (2023), 103341.

[139] Junfeng Tian, Wenjing Xing, and Zhen Li. 2020. BVDetector: A program slice-based binary code vulnerability intelligent detection system. *Information and Software Technology* 123 (2020), 106289.

[140] Zhenzhou Tian, Binhui Tian, Jiajun Lv, Yanping Chen, and Lingwei Chen. 2024. Enhancing vulnerability detection via AST decomposition and neural sub-tree encoding. *Expert Systems with Applications* 238 (2024), 121865.

[141] Tong Wan, Lu Lu, Hao Xu, and Quanyi Zou. 2023. Software vulnerability detection via doc2vec with path representations. In *Proceedings of the 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C'23)*. IEEE, 131–139.

[142] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2020. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security* 16 (2020), 1943–1958.

[143] Mingke Wang, Chuanqi Tao, and Hongjing Guo. 2023. LCVD: Loop-oriented code vulnerability detection via graph neural network. *Journal of Systems and Software* 202 (2023), 111706.

[144] Qian Wang, Zhengdao Li, Hetong Liang, Xiaowei Pan, Hui Li, Tingting Li, Xiaochen Li, Chenchen Li, and Shikai Guo. 2024. Graph confident learning for software vulnerability detection. *Engineering Applications of Artificial Intelligence* 133 (2024), 108296.

[145] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. 2018. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1267–1293.

[146] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. IEEE, 297–308.

[147] 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 *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*. IEEE, 2249–2261.

[148] Yan Wang, Peng Jia, Xi Peng, Cheng Huang, and Jiayong Liu. 2023. BinVulDet: Detecting vulnerability in binary program via decompiled pseudo code and BiLSTM-attention. *Computers & Security* 125 (2023), 103023.

[149] Laura Wartschinski, Yannic Noller, Thomas Vogel, Timo Kehrer, and Lars Grunske. 2022. VUDENC: Vulnerability detection with deep learning on a natural codebase for Python. *Information and Software Technology* 144 (2022), 106809.

[150] Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M. Zhang, and Qing Liao. 2023. Vulnerability detection with graph simplification and enhanced graph representation learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*. IEEE, 2275–2286.

[151] Xin-Cheng Wen, Cuiyun Gao, Jiaxin Ye, Yichen Li, Zhihong Tian, Yan Jia, and Xuan Wang. 2024. Meta-path based attentional graph learning model for vulnerability detection. *IEEE Transactions on Software Engineering* 50 (2024), 360–375.

[152] Bolun Wu, Futai Zou, Ping Yi, Yue Wu, and Liang Zhang. 2023. SlicedLocator: Code vulnerability locator based on sliced dependence graph. *Computers & Security* 134 (2023), 103469.

[153] Hongjun Wu, Zhuo Zhang, Shangwen Wang, Yan Lei, Bo Lin, Yihao Qin, Haoyu Zhang, and Xiaoguang Mao. 2021. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In *Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE'21)*. 378–389.

[154] Tongshuai Wu, Liwei Chen, Gewangzi Du, Dan Meng, and Gang Shi. 2024. UltraVCS: Ultra-fine-grained variable-based code slicing for automated vulnerability detection. *IEEE Transactions on Information Forensics and Security* 19 (2024), 3986–4000.

[155] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An image-inspired scalable vulnerability detection system. In *Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE'22)*.

[156] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*.

[157] Peng Xiao, Qibin Xiao, Xusheng Zhang, Yumei Wu, and Fengyu Yang. 2024. Vulnerability detection based on enhanced graph representation learning. *IEEE Transactions on Information Forensics and Security* 19 (2024), 5120–5135.

[158] Wei Xiao, Zhengzhang Hou, Tao Wang, Chengxian Zhou, and Chao Pan. 2024. MSGVUL: Multi-semantic integration vulnerability detection based on relational graph convolutional neural networks. *Information and Software Technology* 170 (2024), 107442.

[159] Rongze Xu, Zhanyong Tang, Guixin Ye, Huanting Wang, Xin Ke, Dingyi Fang, and Zheng Wang. 2022. Detecting code vulnerabilities by learning from large-scale open source repositories. *Journal of Information Security and Applications* 69 (2022), 103293.

[160] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

[161] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 359–368.

[162] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. 2011. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX Workshop on Offensive Technologies (WOOT'11)*.

[163] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*. 499–510.

[164] Han Yan, Senlin Luo, Limin Pan, and Yifei Zhang. 2021. HAN-BSVD: A hierarchical attention network for binary software vulnerability detection. *Computers & Security* 108 (2021), 102286.

[165] Hongyu Yang, Haiyun Yang, Liang Zhang, and Xiang Cheng. 2022. Source code vulnerability detection using vulnerability dependency representation graph. In *Proceedings of the 2022 IEEE International Conference on Trust, Security, and Privacy in Computing and Communications (TrustCom'22)*. IEEE, 457–464.

[166] Limin Yang, Xiangxue Li, and Yu Yu. 2017. VulDigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes. In *Proceedings of the 2017 IEEE Global Communications Conference (GLOBECOM'17)*. IEEE, 1–7.

[167] Suan Hsi Yong and Susan Horwitz. 2005. Using static analysis to reduce dynamic analysis overhead. *Formal Methods in System Design* 27 (2005), 313–334.

[168] Xiaozhou You, Hui Li, Han Wang, and Faisal Mehmood. 2023. SmartDT: An effective vulnerability detection system of smart contracts based on deep learning. In *Proceedings of the 2023 IEEE International Conference on Big Data (Big Data'23)*. IEEE, 2369–2376.

[169] Lei Yu, Junyi Lu, Xianglong Liu, Li Yang, Fengjun Zhang, and Jiajia Ma. 2023. PSCVFinder: A prompt-tuning based framework for smart contract vulnerability detection. In *Proceedings of the 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE'23)*. IEEE, 556–567.

[170] Bin Yuan, Yifan Lu, Yilin Fang, Yueming Wu, Deqing Zou, Zhen Li, Zhi Li, and Hai Jin. 2023. Enhancing deep learning-based vulnerability detection by building behavior graph model. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*. IEEE, 2262–2274.

[171] Dawei Yuan, Xiaohui Wang, Yao Li, and Tao Zhang. 2023. Optimizing smart contract vulnerability detection via multi-modality code and entropy embedding. *Journal of Systems and Software* 202 (2023), 111699.

[172] Cheng Zeng, Chun Ying Zhou, Sheng Kai Lv, Peng He, and Jie Huang. 2021. GCN2defect: Graph convolutional networks for SMOTETomek-based software defect prediction. In *Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE'21)*. IEEE, 69–79.

[173] Peng Zeng, Guanjun Lin, Lei Pan, Yonghang Tai, and Jun Zhang. 2020. Software vulnerability analysis and discovery using deep learning techniques: A survey. *IEEE Access* 8 (2020), 197158–197172.

[174] Chunyong Zhang, Bin Liu, Yang Xin, and Liangwei Yao. 2023. CPVD: Cross project vulnerability detection based on graph attention network and domain adaptation. *IEEE Transactions on Software Engineering* 49, 8 (2023), 4152–4168.

[175] Chunyong Zhang, Tianxiang Yu, Bin Liu, and Yang Xin. 2024. Vulnerability detection based on federated learning. *Information and Software Technology* 167 (2024), 107371.

[176] Hengyan Zhang, Weizhe Zhang, Yuming Feng, and Yang Liu. 2023. SVScanner: Detecting smart contract vulnerabilities via deep semantic extraction. *Journal of Information Security and Applications* 75 (2023), 103484.

[177] Zhuo Zhang, Yan Lei, Meng Yan, Yue Yu, Jiachi Chen, Shangwen Wang, and Xiaoguang Mao. 2022. Reentrancy vulnerability detection and localization: A deep learning based two-phase approach. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22)*. 1–13.

[178] Zixian Zhen, Xiangfu Zhao, Jinkai Zhang, Yichen Wang, and Haiyue Chen. 2024. DA-GNN: A smart contract vulnerability detection method based on dual attention graph neural network. *Computer Networks* 242 (2024), 110238.

[179] Weining Zheng, Yuan Jiang, and Xiaohong Su. 2021. Vu1SPG: Vulnerability detection based on slice property graph representation learning. In *Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE'21)*. IEEE, 457–467.

[180] Weining Zheng, Yuan Jiang, and Xiaohong Su. 2021. Vu1SPG: Vulnerability detection based on slice property graph representation learning. In *Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE'21)*. IEEE, 457–467.

[181] Zhangqi Zheng, Yongshan Liu, Bing Zhang, Xinqian Liu, Hongyan He, and Xiang Gong. 2023. A multitype software buffer overflow vulnerability prediction method based on a software graph structure and a self-attentive graph neural network. *Information and Software Technology* 160 (2023), 107246.

[182] Kuo Zhou, Jing Huang, Honggui Han, Bei Gong, Ao Xiong, Wei Wang, and Qihui Wu. 2023. Smart contracts vulnerability detection model based on adversarial multi-task learning. *Journal of Information Security and Applications* 77 (2023), 103555.

[183] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. *Devign*: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS'19)*. 1–11.

[184] Yaqin Zhou and Asankhaya Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th ACM Joint Meeting on Foundations of Software Engineering (FSE'17)*. 914–919.

[185] Huijuan Zhu, Kaixuan Yang, Liangmin Wang, Zhicheng Xu, and Victor S. Sheng. 2023. GraBit: A sequential model-based framework for smart contract vulnerability detection. In *Proceedings of the 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE'23)*. IEEE, 568–577.

[186] Weiyuan Zhuang, Hao Wang, and Xiaofang Zhang. 2022. Just-in-time defect prediction based on AST change embedding. *Knowledge-Based Systems* 248 (2022), 108852.

[187] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2020. Smart contract vulnerability detection using graph neural network. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI'20)*. 3283–3290.

[188] Deqing Zou, Yutao Hu, Wenke Li, Yueming Wu, Haojun Zhao, and Hai Jin. 2022. mVulPreter: A multi-granularity vulnerability detection system with interpretations. *IEEE Transactions on Dependable and Secure Computing*. Early Access, August 22, 2022.

[189] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. muVulDeePecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2019), 2224–2236.

[190] Deqing Zou, Yawei Zhu, Shouhuai Xu, Zhen Li, Hai Jin, and Hengkai Ye. 2021. Interpreting deep learning-based vulnerability detector predictions based on heuristic searching. *ACM Transactions on Software Engineering and Methodology* 30, 2 (2021), 1–31.

Received 5 July 2023; revised 31 August 2024; accepted 12 September 2024