Neural Computing and Applications (2023) 35:1279–1301
https://doi.org/10.1007/s00521-022-08046-y

REVIEW

The application of neural network for software vulnerability detection: a review

Yuhui Zhu[1,3] · Guanjun Lin[2] · Lipeng Song[3] · Jun Zhang[4]

Received: 5 November 2021 / Accepted: 7 November 2022 / Published online: 27 November 2022

Abstract

To date, being benefited from the ability of automated feature extraction and the performance of software vulnerability identification, deep learning techniques have attracted extensive attention in data-driven software vulnerability detection. Many methods based on deep learning have been proposed to speed up and intelligentize the process of vulnerability identification. Although these methods have shown significant advantages over traditional machine learning ones, there is an apparent gap between the deep learning-based detection systems and human experts in understanding potentially vulnerable code semantics. In some real-world vulnerability prediction scenarios, the performance of deep learning-based methods drops by more than 50% compared to these methods' performance in experimental scenarios. We define this phenomenon as the perception gap by examining and reviewing the early software vulnerability detection approaches. Then, from the perspective of the perception gap, this paper profoundly explores the current software vulnerability detection methods and how existing solutions endeavor to narrow the perception gap and push forward the development of the field of interest. Finally, we summarize the challenges of this new field and discuss the possible future.

Keywords Deep neural network · Deep learning · Machine learning · Vulnerability detection

1 Introduction

With the popularity of mobile devices and computer networks, software systems have played a critical role in all aspects of our society. Meanwhile, software vulnerabilities arising from software significantly impact businesses and people's lives [1, 2]. A recent study has pointed out that the Internet suffered from nearly 800 million malware attacks in the second quarter of 2018, which reached a high record [3]. Moreover, most of the attacks can be attributed to vulnerabilities in software. Additionally, the number of vulnerabilities reported publicly to the Common Vulnerabilities and Exposures database (CVE) has increased annually, with the number reported in 2021 hitting 20,000.

Identifying vulnerabilities before deploying software is an effective solution to reduce potential losses caused by malicious attacks [4]. To identify vulnerabilities effectively, researchers have proposed many detection methods which can be categorized into static, dynamic, and hybrid techniques [5]. Static techniques, such as rule/template-based analysis [6], static symbolic execution [7, 8], and code similarity detection [9–11], analyze given programs based on source code, and the high false-positive rate is a

Yuhui Zhu and Guanjun Lin made an equal contribution to the paper.

✉ Lipeng Song
slp880@sdu.edu.cn

Yuhui Zhu
zhu425066454@gmail.com

Guanjun Lin
guanjun.lin@fjsmu.edu.cn

Jun Zhang
junzhang@swin.edu.au

1 School of Data Science and Technology, North University of China, Taiyuan 030051, China

2 School of Information Engineering, Sanming University, Sanming 365004, Fujian Province, China

3 The School of Mechanical, Electrical and Information EngineeringSchool of Mechanical, Electrical & Information Engineering, Shandong University, Weihai 264209, China

4 School of Science, Computing and Engineering Technologies, Swinburne University of Technology, Melbourne, VIC 3122, Australia

significant limitation of these techniques. Dynamic techniques analyze given programs by generating specific input data, often accompanied by low code coverage [12]. Finally, a given program is analyzed with a mixture of static and dynamic techniques in hybrid techniques [5]. However, they also suffer from the limitations of both approaches [13]. These methods effectively improve the efficiency of vulnerability detection to a certain extent. However, due to the significant growth of software codes in size and complexity, these solutions fail to satisfy the increasing need for more efficient and effective detection due to the high demand for manual analysis [14].

In order to improve the efficiency and effectiveness of vulnerability detection techniques, many pattern recognition and machine learning (ML) techniques have been widely used to build defect prediction models [15–17]. Based on pioneer studies, researchers have selected source-code-based features such as function call [11], software complexity measurement, and code change [18] as indicators to predict the vulnerable code fragments based on ML approaches. However, ML-based techniques still require experts to define indicators explicitly [19, 20]. Furthermore, it is difficult to reflect on the complex and variable vulnerability patterns and discover new vulnerabilities using these indicators.

The emerging deep learning (DL) approaches offer new potential for software vulnerability detection (SVD). On the one hand, DL approaches could extract high-level features automatically, relieving experts from tedious feature engineering tasks [21]. On the other hand, the DL approaches usually have better generalization abilities and can improve detection performance. [22, 23]. It could discover latent features that a human expert might never consider including and represent them in high-dimensional space [24, 25]. Therefore, DL has found its applications in SVD, and the DL-based SVD has become a promising field.

Researchers' goal is to make the vulnerability detection system like an experienced expert to judge whether a piece of code is vulnerable so that developers can be assisted in identifying and fixing vulnerabilities more efficiently. The SVD methods based on DL are capable of reasoning and understanding code semantics, which shows the possibility of achieving this goal. Researchers are presently pursuing the potential of the DL approach to increase the accuracy of SVD, as indicated by the growing number of scholarly articles (see Fig. 1). The success of DL for SVD expresses the need for an inclusive review of the literature for successive researchers to continue to contribute to this promising field.

How is this survey different from others? Although several related surveys have been published in recent years on SVD, few of them have deeply analyzed how researchers allow full play to the advantages of DL techniques in this field. Existing reviews failed to cover the most recent studies that revealed new research directions because the field of DL for SVD is rapidly advancing. Table 1 shows the difference in scopes, focused topics, and the number of reviewed DL-based SVD works of those reviews. It can be seen from the table that *the existing surveys rarely concentrate their work on DL-based SVD*. They restrict their surveys to conventional ML-based approaches [27, 5, 31] or traditional SVD techniques, including static and dynamic analysis [26, 28–30]. Therefore, there is a need to reveal the trend and progress of the application of DL for SVD. The survey conducted by Lin et al. [4] is the closest to our work as they have specifically reviewed DL publications in vulnerability detection and examined how to facilitate the understanding of code semantics by DL techniques. However, due to the rapid development of this research field, their work, which reviewed only 19 relevant papers, can not cover many critical recent advancements. Along with DL-based SVD's rapid ascent in popularity, a comprehensive review inclusive of more papers would be of great value for the researchers to gain deeper insight into these advancements. Hence, in our work, we review 48 studies that apply DL for vulnerability detection to provide a comprehensive picture of the 5 year advancements in this field of research. Finally, we provide a comprehensive discussion on future directions and challenges for this new area of research.

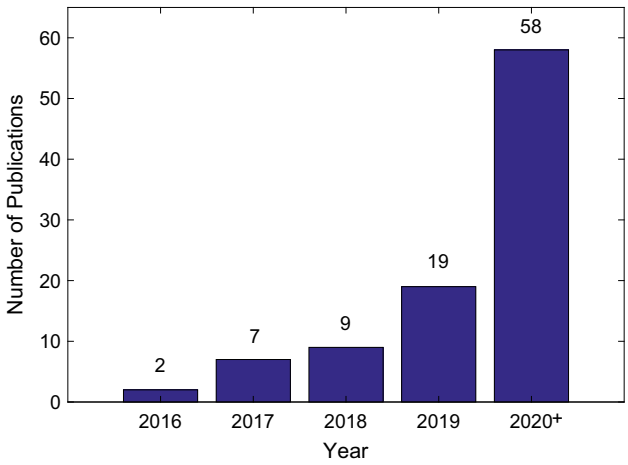

Fig. 1 Recent growth in the number of DL-based SVD scientific publications

How did we select the papers? DataBase systems and Logic Programming (DBLP) and Google Scholar are the two primary databases containing papers in the computer field. We search for the relevant papers in these two databases using several keywords, including vulnerability/faulty detection, bug, source code, and DL. Furthermore, the paper selection focuses on English publications

Table 1 Summary of related survey

Reference	Technology scope	Topic Focus and Taxonomy	Reviewed DL-based SVD works
Liu et al. [26]	SVD including static analysis, Fuzzing and penetration testing	A brief review of SVD involve code analysis and ML-based techniques	None [t]
Malhotra et al. [27]	ML-based SVD	Systematic review of ML-based SVD techniques	None
Ghaffarian et al. [5]	ML-based SVD	An extensive review of SVD that utilize ML and data-mining techniques	None
Ji et al. [28]	SVD, vulnerability exploitation and patching techniques	Extensive review of the studies of automatic SVD, exploitation, and patching techniques	1
Allamanis et al. [29]	Big code analysis including SVD	A systematic review of Big Code based on ML from the difference of programming languages with natural languages	1
Shahriari et al. [30]	SVD including static analysis, Fuzzing and hybird methods	Extensive surevey of SVD including namely testing, static analysis, and hybrid analysis	None
Jie [31]	ML-based SVD	Review of ML-based SVD techniques	None
Lin et al. [4]	DL-based SVD	An in-depth review of the studies focusing on DL techniques and examined how to facilitate the understanding of code semantics by DL techniques	19
This survey	DL-based SVD	Review of the application of DL to SVD techniques and Explain how to combine human cognitive ability with DL techniques to improve the efficiency of vulnerability detection	48 [b]

from high-quality journals and conferences. It ensures that the selected papers have promising innovations in SVD. Our work retrieved more than 90 DL-based SVD researches published in recent five years. However, we finally selected 48, only those published with noteworthy contributions to the field.

Contributions of this survey Our research intends to comprehensively assess the literature on DL-based SVD and demonstrate recent advances in the field. In addition, it could serve as a guide for the researchers to breathe how DL techniques are applied for solving different aspects of the SVD problems and know the limitations and future directions of this area. We summarise the significant contributions as follows:

(1) We review 48 recent DL-based SVD studies, presenting a research trend in this active field.
(2) We identify a gap between human understanding and vulnerability detection systems, defined as the perception gap. From the perspective of the perception gap, we categorize how existing studies contribute to bridging this gap by combining the human experience with the advancement of DL.
(3) We compare the papers published in 2016–2017 with those published later and discuss current limitations, challenges, and opportunities of DL-based SVD, covering recent achievements. Moreover, we conducted several experiments based on real-world data sets to clarify our views.

The paper is organized as follows. In Sect. 2, we first analyze the shortcomings of current SVD methods, then review the DL-based SVD methods in the early stage, and summarize its primary process. Section 3 reviews the remaining papers and categorizes them according to their research contents, focusing on analyzing how the researchers bridge the gap between DL-based SVD methods and human understanding from the proposed methods and solutions. Section 4 elaborates on lessons learned and future research directions in the field of DL-based SVD. Finally, Sect. 5 concludes the paper.

2 The gap between human understanding and vulnerability detection systems

2.1 The dilemma and potential of vulnerability detection systems

Designing a vulnerability detection system is to find vulnerabilities hidden in software and assist in completing the code inspection [32]. Moreover, the data-driven vulnerability detection system needs to analyze code from the level of code semantics and syntax. However, the structure of vulnerability codes is very complex, which could be evidenced by CVE-2017-11176. The diff file of CVE-2017-11,176 is shown in Fig. 2.

```
diff --git a/ipc/mqueue.c b/ipc/mqueue.c
index c9ff943f19abc..eb1391b52c6f8 100644
--- a/ipc/mqueue.c
+++ b/ipc/mqueue.c
@@ -1270,8 +1270,10 @@ retry:

                timeo = MAX_SCHEDULE_TIMEOUT;
                ret = netlink_attachskb(sock, nc, &timeo, NULL);
-               if (ret == 1)
+               if (ret == 1) {
+                    sock = NULL;
                     goto retry;
+               }
                if (ret) {
                     sock = NULL;
                     nc = NULL;
```

Fig. 2 The diff file of CVE-2017-11,176

The vulnerability CVE-2017-11,176 is located in the Linux kernel. In this function, message queuing allows asynchronous event notification. When a message is placed in an empty queue, the message queue allows for a signal or a thread to start. This asynchronous event notification calls *mq_notify* function. And *mq_notify* creates or removes asynchronous notifications for the specified queue. Because *mq_notify*, the notify function, did not set the socket pointer to null when entering the retry process, which may cause a use after free (UAF) vulnerability. There are more than 20 functions related to the trigger process of the vulnerability, shown in Fig. 3, and the code that needs to be modified to fix the vulnerability is not wholly consistent with the code that caused the vulnerability.

It is difficult for the vulnerability detection system to understand the causes of such complex vulnerabilities, even with an expressive and specially-crafted model and sufficient training data. For a human expert, gained experience in code inspection, knowledge of the program, and understanding of the programming language are also required to identify such vulnerability. Therefore, there is a gap between human understanding and vulnerability detection systems.

Many studies have mentioned this problem and pointed out that the key is that manual features can not ultimately present code semantics and syntax information [33–35]. Therefore, a previous review [4] presented a concept of the semantic gap, which is defined as: *"The semantic gap is the lack of coincidence between the abstract semantic meanings of a vulnerability that a practitioner can understand and the obtained semantics that an ML algorithm can learn"*. The researchers believe that DL-based vulnerability detection systems can narrow the semantic gap by learning intricate patterns and high-level representations that reveal the code semantics of software codes [36, 37].

The gap between human understanding and vulnerability detection systems is reflected in the understanding of code semantics and the cognition of the whole vulnerability detection task: People's understanding of the task is to find vulnerabilities. However, the detection systems can only find similar codes to the provided vulnerability samples for training but cannot analyze whether these codes are vulnerabilities.

In addition, some vulnerabilities may be triggered in specific conditions. The same piece of code may be identified as vulnerable when processing some particular tasks. Thus, experts can find vulnerabilities by analyzing differences between the process of codes execution with the actual requirements. However, the current data-driven vulnerability detection methods lack these bits of knowledge.

Therefore, the gap between human understanding and vulnerability detection systems cannot be covered entirely by the definition of the semantic gap because this definition can only cover the gap between the detection system and

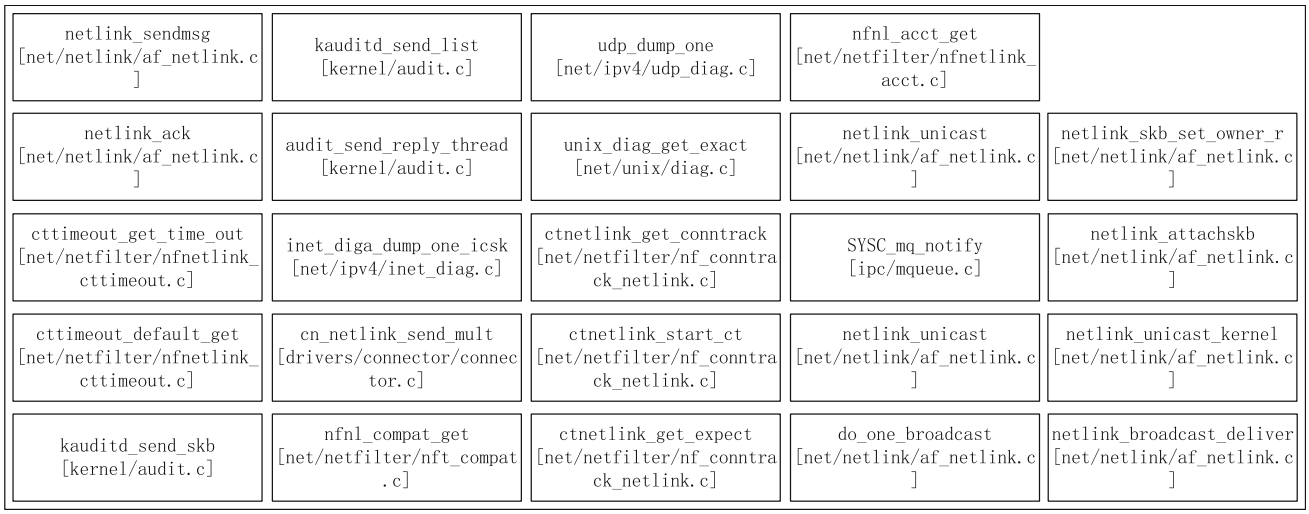

Fig. 3 Involved function of CVE-2017-11,176

human experts in the comprehension of code semantics without covering the gap in the comprehension of code relevant information. Section 2.2 will review preliminary researches in DL-based SVD. On this foundation, we will further analyze the perception gap in Sect. 2.3.

2.2 Preliminary researches in DL-based SVD

Because the application of the DL model does not rely on expert-defined features, it has been favored in various fields [38–40], such as speech recognition [41, 42], image recognition [43, 44], and machine translation [45, 46]. Furthermore, DL has become the basis of the most advanced artificial intelligence applications [47]. So far, Deep Neural Networks (DNN) have also been applied to SVD, and their detection performance is encouraging. In this section, we would like to discuss how the preliminary studies apply DL to vulnerability detection.

(1) *Summary of recent works*: To our knowledge, the first study that utilized DNN for SVD was proposed in [32]. Subsequently, some SVD methods based on DL were published within two years [22, 32, 34, 48–51], forming a relatively complete process.

In the first study [32], the authors adopted a deep belief network (DBN) for detecting bugs and defects in Java source code. Since Abstract Syntax Trees (ASTs) provide a structured representation of the source code function and reserve more syntactic information than source code, ASTs represent the semantic and syntactic information hidden in the source code. Their ASTs contain function nodes, declaration nodes, and control-flow nodes in three types of nodes. The reason why they excluded other AST nodes was to prevent diluting the importance of other nodes. Then, the authors adopted the method proposed in [53] to reduce the noise in the data set. To input ASTs into the DBN network, the author maps AST nodes to tokens and uses the method proposed in [54] to limit the input tokens between 0 and 1. This paper adopted a generative graphical model DBN to learn vulnerability code representation from a labeled data set. DBN contains one input layer and several hidden layers, and the top layer is the output layer. Each layer consists of several stochastic nodes. The author assumes that this multi-layer structure can enable DBN to reconstruct the semantics and content of input data with high probability and learn the representation of vulnerabilities. However, this paper's methods only work at the file-level, and it can not pinpoint the vulnerabilities related to code lines.

In research [49], a Convolutional Neural Network (CNN) was adopted to generate semantic and structural features of the source codes. Moreover, the features were combined with 20 traditional features, which were extracted by Jureczko et al. [55] to distinguish whether there was a vulnerability in the file. The AST they used was the same as that of [32]. Thus each source file was represented by a token vector. Subsequently, they adopted a CNN to extract semantic and structural features from source codes. Moreover, they combined the extracted features with traditional features. Then, the authors applied a Logistic Regression classifier [56] to judge whether an input test code was buggy.

A function-level AST-based approach was proposed in [22]. Compared with file-level vulnerability detection, it can pinpoint the key code better. The authors collected more than 6000 labeled functions from three open-source projects. For cross-project vulnerability discovery, a Bi-directional Long Short-Term Memory (BiLSTM) network was adopted to learn code representations in this study. Since ASTs and the source codes lack control flow information and can not reflect the control dependencies, a method, CNNs over Control Flow Graphs (CFGs) for SVD, was proposed in [34]. The authors collected four real-world data sets from a popular programming contest site CodeChef[1] for conducting experiments.

Previous studies used source code as training data, with expensive manual marking consumption. An approach for predicting memory corruption vulnerabilities was proposed in [52], and the authors extracted features from both static and dynamic analysis. The authors thought that the usage patterns of the C library functions could exhibit the memory corruption vulnerabilities, and they extracted the call sequences/traces from a set of call sequences associated with the standard C library functions and the monitor of programs' execution for a limited period. Later, a study that also used function call sequences as features was proposed in [51]. Similarly, they used the method proposed in [52] to get function call sequences. Nevertheless, the difference was that they used a popular multi-purpose fuzzer zzuf[2] to detect unexpected behavior to acquire the label of every program. The disadvantage of this work was that the quality of the data set could not be guaranteed, and the data label method is only suitable for some types of vulnerabilities.

(2) *Discussion*: Given existing research achievements, compared with ML, even a simple DL model such as Multilayer Perceptron (MLP) in [32] has a better detection effect in various SVD tasks, which benefits from learning high-level features or representations with more complexity and abstraction. Moreover, DL techniques allow the detection systems to capture code semantics, understand contextual code dependencies, and automatically extract high-dimensional features that better reflect vulnerabilities' essence. With these capabilities, although the constructed

[1] https://www.codechef.com/.

[2] http://caca.zoy.org/wiki/zzuf.

SVD systems are not enough to dig out various vulnerabilities in source codes like human experts, DL-based SVD systems have had a better performance than all other types of data-driven vulnerability detection systems. At present, DL-based SVD technology is still in its infancy. DL technology is expected to realize automatic and intelligent vulnerability mining with continuous improvement.

2.3 Vulnerability perception gap

Although researchers have applied various SVD methods based on DL, several recent studies have demonstrated that the accuracy of DL-based SVD could be up to 90% at detecting vulnerabilities in experimental scenarios. Nevertheless, in some actual detection environments, their performance dropped by more than 50% [57]. The current DL-based SVD methods are still limited in their applications.

Hence, there is a gap between human understanding and vulnerability detection systems. We call it the perception gap and define it as follows:

The perception gap is a lack of consistency between practitioners' cognition of object code and decision-making of the DL model.

This gap has two aspects: on the one hand, due to the black-box nature of many DL-based methods for SVD, people can not comprehensively understand the decision process and reasons for the DL-based detection system; on the other hand, there are still shortcomings in the DL system compared to human experts.

The lack of sufficient data is one of the most critical reasons [4, 58]. DL methods usually need large data sets, especially for complex tasks like vulnerability detection. However, the known public data set, SARD,[3] is not collected for vulnerability detection. Therefore, the lack of sufficient data for training leads to the DL method not being able to extract complete vulnerability features. Moreover, due to not understanding the detection task like a human, the in-depth learning method needs to be further optimized to achieve better results in various scenarios. In addition, the lack of fine-grained and interpretable DL-based SVD systems is also the reason for this gap.

How can this gap be bridged? If an ordinary person wants to be an experienced expert, an overall and local understanding of tasks of interest is essential and it is also true for DL-based SVD methods. Therefore, researchers integrate their understanding of vulnerability detection tasks into the deep learning model by optimizing each step of DL-based SVD. Therefore, the detection system can understand vulnerability detection tasks like human experts.

[3] https://samate.nist.gov/index.php/SARD.html.

As shown in Fig. 4, DL-based SVD methods have formed a relatively complete process, including data collection, code representation, model building, and evaluation/test.

Data collection is to collect labeled data sets for training neural networks. In this part, the critical point is the grain of high quality labeled data. In general, fine-grained vulnerability labels can better locate the code. However, the size of current vulnerability data is relatively small. Therefore, the insufficient training data can not meet the requirements.

Feature representation can be divided into code representation and word embedding methods. Code representation methods aim to express the semantic and grammatical information that the source codes miss, and word embedding methods aim to turn code representations into vectors that neural networks can process. For most neural networks, the code representations can be well input into deep network training, such as Word2vec [59, 60]. However, due to the complex software background and expert experience related to the vulnerabilities, current code representation methods can not completely express the semantic and grammatical information that the source codes miss.

Model building applies or customizes deep neural network models that automatically extract the vulnerable patterns to detect potential vulnerabilities. The model structure determines the learning ability of the model for different types of data. Therefore, a model with better learning ability and is easy to explain is expected.

Evaluation/test is to train and test the built detector in specific application scenarios. The DL model needs to be optimized for different scenarios. The current research mainly focuses on feasibility and theoretical research, and many problems need to be solved in practical application [57].

Therefore, DL-based SVD can better complete the vulnerability detection task by combining the experts' understanding of the vulnerability detection task, which can bridge the perception gap.

3 DL-based SVD for bridging the perception gap

DL has been extensively used in natural language processing, such as machine translation and language understanding, and is also suitable for code semantic analysis, which can help human experts screen possible vulnerability codes. It prompted many researchers to follow up on the DL-based SVD methods to improve the vulnerability detection ability.

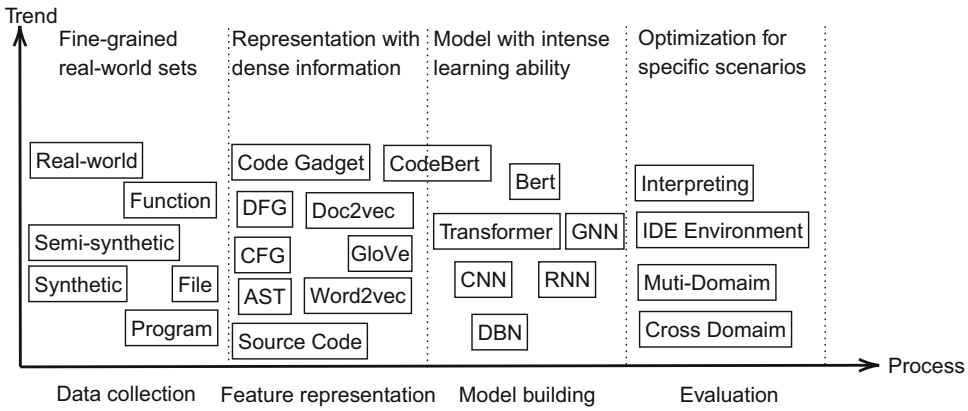

Fig. 4 Overview of DL-based SVD process

3.1 Human experience facilitating DL-based SVD

Section 2.2 has introduced early DL-based research, mainly focusing on realizing a complete detection system. This section will focus on how researchers optimize the application of DL models in vulnerability detection.

The origin of the perception gap is that the deep learning method can not obtain the relevant knowledge of software background and expert experience, and this knowledge can not extract from the training data [61]. However, relevant knowledge is essential for vulnerability mining. Therefore, researchers analyze and extract related knowledge and try their best to input this knowledge into the DL-based SVD models. Thus, this section divides the reviewed papers into four main optimization directions for DL-based SVD. Next, we will introduce the four directions.

Improvements in the quality of data sets: It mainly aims to optimize data acquisition, labeling, and processing methods, which objectively improve the quality of data sets and reduce the demand for computing power. The deep learning models are able to accurately extract vulnerability features with improved data quality and more accurate labeling.

More suitable feature representation methods: Feature representation can be divided into code representation and word embedding methods. In this research direction, the researchers select the code representation and word embedding methods ideal for detecting different types of vulnerabilities according to their experience. Combined with the mighty computing power of the DL model, these code representation methods could further improve the feature extraction ability of DL methods. At the same time, a suitable word embedding method can better retain the semantics information of code representation.

Neural networks with improved learning ability: On the one hand, researchers can improve the learning ability of neural networks by optimizing the structure of networks or increasing the scale of networks. On the other hand, feature input neural networks can also be optimized to retain more semantic and grammatical features.

Optimization for specific scenarios: Optimize for the specific problems encountered in the actual detection environment, such as the software to be tested does not allow a view of the code, and the computing power of the detection environment is insufficient.

The rationale behind the proposed categorization is as follows: Integrating more expert experience into each step of DL-based SVD methods can narrow the perception gap between DL-based SVD methods and human experts. The following sections will review some essential works to illustrate how researchers bridge the perception gap in different directions.

3.2 Improvements in the quality of data sets

This section shows essential works that optimize data sets' quality from three aspects: data source, granularity, and label quality.

(1) *Summary of recent works*: In [62], A new data labeling method is proposed, and the authors collected a extensive data set with 12 million source code based on this method. They used three open-source static analyzers, Clang,[4] Cppcheck,[5] and Flawfinder.[6] However, the data label may not be consistent with the actual label, and the method can not guarantee the quality of data.

A statement-level data generator method was proposed for detecting buffer overflow vulnerabilities in [63]. Compared with the research [48], it provided a synthetic code generator that could generate codes that can be compiled normally, and their data sets have control flow structures and code line numbers. The author used the libclang interface to split the code files into statement-level codes.

[4] https://clang.llvm.org.

[5] http://cppcheck.sourceforge.net.

[6] https://dwheeler.com/flawfinder.

Then a statement-level based detector called VulDeeLocator was developed by [64]. The authors used intermediate codes as the program's representation to detect vulnerability at the slice-level. The authors obtained intermediate codes by the Lower Level Virtual Machine (LLVM). The extraction and segmentation of statement codes needed a tool, and the location of the vulnerability codes was not precise enough. Then, a line-level classification method called Vulcan was proposed in [37]. The authors investigated the problem of classifying a line of the program as containing a vulnerability or not using ML.

Then, An extension method of [22] was proposed in [65]. In this work, the authors used the real-world data set collected from GitHub and proposed a novel fuzzy-oversampling method to address the non-vulnerable data insufficient issue. To provide sufficient real data, the authors provided more labeled code from nine different open-source software in research [67]. The granularity of the data set covers function-level and file-level. At the function-level, it contains 1,471 labeled vulnerable and 59,297 labeled non-vulnerable source code functions. And at the file-level, it contains 1,320 vulnerable and 4,460 non-vulnerable. The experiment results were conducted on the proposed real-world data set and SARD data set with different network structures. In [67] a deep domain adaptation method was proposed to solve the problem of lacking enough labeled data. To overcome the lack of labeled vulnerability data, the authors adopted a semi-supervised variant to fully utilize the unlabeled target data's information by treating the unlabeled target data as the unlabeled component in semi-supervised learning. Subsequently, they use spectral graphs [72] to represent the geometry of data and optimize the output results via minimizing the conditional entropy [73] of the source and target distribution.

Research [67] has shown that semi-supervised learning has the potential to alleviate the lack of large-scale labeled data sets effectively. Thus, it can provide more fine-grained labeled data and make the DL model locate vulnerability code more accurately. However, it is equally important to accurately distinguish the type of vulnerability, which can more accurately explain why this code segment contains a vulnerability. A recent study [69] proposed a multi-class vulnerability detection can effectively solve this problem. The source code was converted to token sequences in the processes, and the authors applied an Long Short-Term Memory (LSTM) network to classify vulnerabilities.

Later, a more efficient multi-class vulnerability detection was proposed in [19]. First, they collected a data set containing 116 different types of vulnerabilities and 33,086 test cases. Then, code gadgets containing data dependencies, control dependencies, and the "global" semantics related to possible vulnerabilities were used. And "code attention" was proposed to focus on "localized" information to detect specific vulnerability types.

Previous studies have greatly improved the vulnerability granularity but also made it more difficult to label source codes. In order to solve the labeling problem of fine-grained data, a differential analysis-based approach called D2A was proposed in [70]. The authors built their data set by analyzing version pairs from multiple open source projects. They select bugfix submissions from each project and statically analyze the versions before and after submission. The detected issues, which disappear in the corresponding after-commit version, are likely to be real bugs. They used this method to generate a large labeled data set.

In addition, a data set collection method was proposed in [57]. To obtain the labeled data set, the authors collected the already fixed issues with publicly available patches of open source software, such as Linux Debian Kernel and Chromium. Later, Wang et al. proposed an automatic data labeling method in [71] that can automatically obtain data from GitHub. It further reduced the labor cost of data set collection. They conducted an automatic framework for collecting vulnerable code samples. In this framework, a set of predictive models or experts were used to predict whether a code commits relevant to a code vulnerability. Moreover, the vulnerability code segment can be identified by comparing the different versions of code before and after the code commit.

(2)*Discussion*: There are three main improvements in the quality of data sets. The first is to find a better data source. In Sect. 2.2, most of the data sets of reviewed papers are not from the actual application software or do not use the source code, for example [48, 52, 51]. Compared with the actual software code, synthetic or semi-synthetic codes possess a simple structure. Some synthetic codes even can not be compiled normally. The vulnerability characteristics extracted from them can not meet the actual vulnerability detection requirements. In addition, the granularity of these data is at the program-level or file-level, which can not meet the needs of accurately locating vulnerabilities.

The second is to improve the granularity of data. Studies in this section significantly improve vulnerability labels' accuracy from file-level to function-level or statement-level. At the same time, specific vulnerability types have been marked in some review papers, such as [69, 19]. These have greatly improved the data quality.

The last is to improve the marking and cleaning methods of data. More than half of the studies extracted some or all data sets from open source software projects compared to previous studies. They optimized the quality of data labels by manual effort [64–67] or by analyzing the differences in software codes in different versions [70, 57, 71].

3.3 More suitable feature representation methods

This section shows some critical works that optimize feature representation methods from two aspects: code representation and word embedding methods.

(1) *Summary of recent works*: In order to fully extract the semantics of programs, Fan et al. combined static measurement methods with ASTs, in [61]. They believed that ASTs could reflect the original structure of source code and reserve more semantic information. And the authors combined these semantic features with traditional static metrics to improve the performance of SVD. With similar intent, another approach was proposed in [74]. The authors consider two types of complementary features for vulnerability detection. The first was CFGs generated by Clang and LLVM [84], Moreover, the second was based on source codes directly. The authors convert two sets of features to token sequences. Then, they converted the generated two types of token sequences into vector representations, using a word package model and word2vec [59, 60] model.

Paper [75] proposed a new code representation method by embedding code comments. The author believed that comments of codes could reflect the semantics and functions of source codes. Thus, the semantic features could be extracted from those comments. Mainly because some codes in the real world lack comments, the author's testing modules did not contain comments. In this way, the classifier could cope with the missing comments situation. Therefore, comments were only fed into the trained model during the testing process.

An approach proposed a program representation called "code gadget" for detecting vulnerabilities in [24]. A code gadget is several lines of code that are semantically related in terms of data dependency or control dependency. Therefore, the code gadget defined could be used to capture the vulnerabilities related to data flow or control flow dependencies. In addition, the authors used a business tool called Checkmarx[7] to generate code gadgets.

In a recent study [76], Li et al. further extended the "code gadget" adopted in [24] and their "code gadget" contains both data dependency and control dependency of code sequences. Furthermore, they implemented the study based on an extended open-source parser Joern [13]. Compared with Checkmarx used in [24], it could accommodate new semantic information of programs.

Then, a name-based bug detection approach for detecting JavaScript bugs was proposed in [77]. It detects accidentally swapped function arguments, incorrect binary operators, and incorrect operands in binary operations. The authors think that the names of variables and functions contain helpful information for the three types of vulnerabilities. Similarly, a name-based vulnerability detector was also used in C/C++ and Python programs' vulnerability detection in [78]. The authors thought function names contain important semantic features to distinguish vulnerability functions in source code. However, the function names usually can not provide enough information, and the detector can not accurately locate the vulnerability.

The reviewed studies have shown that ASTs and CFGs effectively represent the semantic and syntactic information hidden in the source code. Later, a representation method that merged ASTs, CFGs, Data Flow Graph (DFGs), and Program Dependence Graphs (PDGs) was proposed in [35]. Data and control dependencies are made clear in a representation known as PDG, which uses graph notation. These dependencies are considered during the dependent analysis phase of compiler optimization, which improves parallelism and uses many cores. The authors called it to code property graph (CPG) and stored it with a joint data structure [13]. Then, an Intermediate Representation (IR) based method was proposed in [79]. The authors thought that as an intermediate code representation containing data and control information, IR could extract vulnerability characteristics in different programming languages.

In addition, an automated and intelligent vulnerability detection method was proposed in [80], and the minimum token sequences representation was used for code representation. The minimum token sequences representation can ensure that more information is input into the neural network and improve long codes' detection ability.

Although many code representation methods have been adopted in vulnerability detection, few studies pay attention to the performance of code representations. Later, an evaluation of vulnerability detection performance on code representations was proposed in [81]. To evaluate the performance of different code representations, the authors proposed a DL framework consisting of 3 DNNs in conjunction with five different representations. The framework contains ASTs, Code Gadgets (CGs), Semantics-based Vulnerability Candidates (SeVCs), Lexed Code Representations (LCRs), and Composite Code Representations (CCRs). Their experiments concluded that the CCRs had the best overall improvement.

Different from the previously reviewed papers, a Kernel Based Extreme Learning Machine (KELM) model that focuses on the optimization of vector representation was proposed in [82]. This paper adopted a multi-level word embedding method to represent the features of code structure better. Specifically, the authors obtained the symbolic representation of the source code related to the vulnerability through three kinds of symbolization and

[7] https://www.checkmarx.com.

Table 2 Reviewed studies which published in 2016–2017

Reference	Data source	Feature / Representation	Neural network model	Detection granularity	Labels obtained from & Provided by
Wang et al. [32]	The PROMISE defect data sets	ASTs	DBN	File-level	The PROMISE [t]
Pradel and Sen [50]	JavaScript	ASTs	Feed-forward network	Statement-level	Automatically generated
Choo et al. [48]	The synthetic data set(CJOC-bAbI)	Generated from source code	Memory network	Statement-level	Automatically generated
Li et al. [49]	The PROMISE defect data sets	ASTs	CNN	File-level	Automatically generated
Lin et al. [22]	Three open source project from Github written in C language	ASTs	BiLSTM	Function-level	Mannual effort
Viet Phan et al. [34]	CodeChef	CFGs	CNN	File-level	Automatically generated
Grieco et al. [52]	Debian program binaries	Static & dynamic call of sequences of C library functions	MLP	Program-level	Debian bug tracker & a simple fuzzer
Wu et al. [51]	32-bit Linux programs	Dynamic call of sequences of C library functions	CNN, LSTM, CNN-LSTM and FCN	Program-level	A fuzzer named zzuf [b]

Table 3 Reviewed studies in Sect. 3.2

Reference	Data source	Feature / Representation	Neural network model	Detection granularity	Labels obtained from & Provided by
Russell et al. [62]	The Juliet Test Suite, Debian programs & open source project from Github	Lexed source code with reduced vocabulary size	RNN/CNN	Function-level	Combining the Clang, Cppcheck & Flawfinder with manual evalution [t]
Sestili D. et al. [63]	CJOC-bAbI & s-bAbI & the buffer overflow test cases from Juliet Test Suite	Generated from source code	Memory network	Statement-level	Automated & the JulietTest Suite
Li et al. [64]	The Juliet Test Suite & open source project from Github	Intermediate code	Bi-RNN-vdl	Statement-level	Automated & mannual effort
Srikant et al. [37]	Ethereum	Control and data dependencies ASTs	BiLSTM with an attention mechanism	Line-level	Automatically generated
Liu et al. [65]	Three open source project from Github written in C language	ASTs	BiLSTM	Function-level	Mannual effort
Lin et al. [66]	Nine open source project from Github written in C language	Source code	DNN Text-CNN and four RNN variants	File-level and function-level	Mannual effort
Nguyen et al. [67]	Six open source project from Github written in C language in [68]	Statements in source code	Bidrectional RNN	Function-level	Mannual effort
Saccente et al. [69]	The Juliet Test Suite	Token sequences	BiLSTM	File-level	Automated
et al. [19]	The Juliet Test Suite & open source project from Github	Code gadgets	Buliding-block LSTM	Code gagets(consisting of multiple statements)	Automated & mannual effort
Zheng et al. [70]	Six open source project from Github	25 manually extracted features	13 well-known ML models	Function-level	Automated & mannual effort
Chakraborty et al. [57]	Two open source project	CPGs	GNN	Function-level	Automatically generated
Wang et al. [71]	Open source project from Github	ASTs+PCDGs	GNN	Function-level	Automatically generated [b]

Table 4 Reviewed studies in Sect. 3.3

Reference	Data source	Feature / Representation	Neural network model	Detection granularity	Labels obtained from & Provided by
Fan et al. [61]	Java projects in Apache	ASTs + static metrics	BiLSTM with an attention mechanism	File-level	Automatically generated [t]
Harer et al. [74]	Debian programs & open source projects from Github	CFGs & source codes	CNN	Function-level	The Clang static analyzer
Huo et al. [75]	The PROMISE defect data sets	source code with embedding comments	CAP-CNN	File-level	The PROMISE
Li et al. [24]	The Juliet Test Suite & open source project from Github	Code gadgets	BiLSTM	Slice-level	Automated & mannual effort
Li et al. [76]	The Juliet Test Suite	Code gadgets	MLP CNN LSTM GRU BiLSTM BGRU	Slice-level	Automated & mannual effort
Pradel et al. [77]	Open-source real-world JavaScript code	ASTs	A feedforward neural network	File-level	Automatically generated
Li et al. [78]	Two open source project from Github	Function names	BiLSTM	Function-level	Automated & mannual effort
Wang et al. [35]	The Juliet Test Suite	ASTs, CFGs, DFGs	CNN & RNN	File-level	Automatically generated
Zaharia et al. [79]	The Juliet Test Suite	IRs	MLP	Function-level	Automatically generated
Li et al. [80]	The Juliet Test Suite	Minimum token sequences	CNN	Function-level	Automatically generated
Zheng et al. [81]	Data sets used in five different work	ASTs, CGs, CCRs, SEVCSs, LCRs	BGRU CNN BiLSTM	Function-level	Automated & mannual effort
Tang et al. [82]	The Juliet Test Suite & open source project from Github	Code gadgets	KELM	Slice-level	Automated & mannual effort [b]
Hin et al. [83]	Open source project from Github	PDGs	GNN	Statement-level	Automated & mannual effort [b]

introduced Doc2vec [85] for vector representation. Thus, it can significantly reduce the noise introduced by irrelevant information about vulnerable codes.

In addition, Hin et al. proposed a new deep learning system, LineVD [86], for detecting statement-level vulnerabilities as a node classification problem. LineVD used a transformer-based approach, CodeBERT [83], to encapsulate the raw source code tokens and Graph Neural Network (GNN) to utilize control and data dependencies between statements.

(2)*Discussion*: This section mainly divides the feature representation method's improvement into two parts. On the one hand, it improves the form of code representation. The source codes or tokens extracted from the source codes can be directly used as a code representation method. However, due to various reasons, such as software updates, the programming language contains much redundant information, and the order of the programming language may be different from the actual execution order in computers. These factors may affect the learning efficiency and effect of the deep learning model. Therefore, researchers applied ASTs, obtained from the compilation process of source code, as code representation methods. After compiling, ASTs remove redundant fragments in the source codes and restore the execution order of source codes. Based on ASTs, researchers increased control flow information and data flow information to AST, forming various representation methods, such as Code gadgets [24, 76] and IRs [79]. In addition, studies [35, 61, 74] combined various different representations to input more features into neural networks. These studies have proved that richer representations help to improve the performance of neural networks.

On the other hand, it mainly enhances the embedding methods. The embedding methods can be divided into non-contextual and contextual embedding technology. Non-contextual embedding technology, such as Word2vec [59, 60] and GloVe [87], converts each word in the text

Table 5 Reviewed studies in Sect. 3.4

Reference	Data source	Feature / Representation	Neural network model	Detection granularity	Labels obtained from & Provided by
Lin et al. [68]	Six open source project from Github written in C language	ASTs	BiLSTM	Function-level	Mannual effort [t]
Duan et al. [88]	The Juliet Test Suite	CPGs	CNN with attention	Line-level	Automatically generated
Fan et al. [89]	Severn open source project from Github written in C language	ASTs	BiLSTM with ATTENTION	File-level	Mannual effort
Zhang et al. [90]	Tera	ASTs	Transformer	Function-level	Automatically generated
Ziems et al. [91]	The Juliet Test Suite	Source codes	Bert+BiLSTM	Function-level	The Juliet Test Suite
Tang et al. [92]	The Juliet Test Suite & open source project from Github	Code gadgets	RVFL	Slice-level	Automated & mannual effort
Zhou et al. [14]	Four open source project from Githubs written in C/C++	AST,CFG,DFG and NCS	GGNN	Function-level	Automatically generated
Brauckmann et al. [33]	The data set consists of the sevenbenchmark suites	ASTs, CDFGs	GNN	Function-level	Automatically generated
Cao et al. [93]	Four open source projects from NVD and GitHub written in C/C++	ASTs, CFGs, DFGs	BGNN+CNN	Function-level	Automatically generated [b]

Table 6 Reviewed studies in Sect. 3.5

Reference	Data source	Feature / Representation	Neural network model	Detection granularity	Labels obtained from & Provided by
Lin et al. [36]	Six open source project from Github written in C language & The Juliet Test Suite	ASTs & source codes	BiLSTM	Function-level	Automated & mannual effort [t]
Liu et al. [96]	The Juliet Test Suite & open source project from Github	ASTs & Code gagets	BiLSTM	Function-level	Mannual effort
Sheng et al. [97]	PROMISE repository	ASTs	GAN-CNN	File-level	The PROMISE
Nguyen et al. [98]	Six open source project in [68]	Source code	DUAL-GAN-BiLSTM	Function-level	Automated & mannual effort
Tanwar et al. [99]	CISCO 8.9 million functions	ASTs	Three-layer Neural network	Function-level	Automated
Bui et al. [100]	Open source project from Github	ASTs	TBCNN+ GGNN	Statement-level	Automated [b]

into a separate high-dimensional vector representation. Contextual embedding technology, such as CodeBERT [83], considers the context information and can recognize polysemy and similar terms according to the context. The effects of different word embedding methods such as Doc2vec [82], GloVe, and CodeBERT on vulnerability detection results in the same scenario are compared in research [86]. The results show that context embedding technology such as CodeBERT can effectively improve the results of vulnerability detection.

3.4 Neural network with improved learning ability

This section shows some works that focus on optimizing neural networks for SVD.

(1) *Summary of recent works*: In order to extract more comprehensive features from source code, a method was proposed in [68]. In this study, a global max-pooling layer was used to capture the most critical features of vulnerabilities. Similarly, the authors used the AST of function as

Table 7 Software vulnerability data sets collected by the reviewed studies at the time of writing

Dataset	Used by	Data type	Granularity	Data scale
PROMISE	[32, 49, 75, 97]	Semi-synthetic	Statement-level	Functions from multiple open source projects [t]
CJOC-bAbI	[48, 63]	Synthetic	Statement-level	Training set of 10,000 functions and test set of 4,000 functions
s-bAbI	[63]	Synthetic	Statement-level	38,400 files and 76,549 buffer writes functions
SARD	[19, 24, 35, 36, 62–64, 69, 76, 79, 80, 82, 88, 91, 92, 96]	Semi-synthetic	Function-level	Test cases of 64,099 files
CodeChef	[34]	Semi-synthetic	File-level	Total 47,064 files
Devign	[14]	Real-world	Function-level	Total 48,687 commits, 58,965 graph
Draper	[62]	Real-world	Function-level	Total 12,753,027 functions, 9,706,269 functions from github and 3,046,758 from debian
Lin	[22, 36, 65–68, 98, 101]	Real-world	Function-level	Nine open source project from Github written in C language, 1,471 vulnerable and 59,297 non-vulnerable source code functions
D2A	[70]	Real-world	Trace-level	Total 1,295,623 unique auto-labeler examples and 18,653 unique after-fix examples
REVEAL	[57]	Real-world	Function-level	Total 18,169 programs [b]

Table 8 The number of vulnerable and non-vulnerable functions on three data sets

Dataset	of vul. functions	of total functions
Lin	1471	60,768
REVEAL	2239	22,734
SARD	94,710	158,007

the original feature. They believed that it could better distinguish the semantic information of different sequences in high-dimensional space. However, some vulnerable and non-vulnerable code is hardly distinguishable, resulting in low detection accuracy. Therefore, an attention mechanism was adopted in paper [88] to capture the critical features of the vulnerabilities. And the Code Property Graph (CPG) was used to obtain semantic features in this framework. A data structure called the CPG was created to explore big codebases for examples of programming patterns. These

Table 9 Cross-domain test results of real-world data set Lin and semi-synthetic data set SARD

Source–Target	System	TOP10 Precision (%)	TOP20 Precision (%)	TOP50 Precision (%)	TOP100 Precision (%)	TOP200 Precision (%)
Lin-REVEAL	BiGRU	18.2	23.8	25.5	27.7	28.9
	BiLSTM	36.4	**47.6**	**41.2**	37.6	**36.8**
	TextCNN	**45.5**	33.3	37.3	**38.6**	33.8
SARD-REVEAL	BiGRU	18.2	23.8	21.6	16.8	15.4
	BiLSTM	45.4	38.9	33.3	36.6	27.3
	TextCNN	27.3	19.0	27.5	20.8	20.4

The bold value indicates the best performance in the same performance metric of different methods

Table 10 Test results of real-world data set Lin in function-level and file-level

Dataset	System	TOP10 Precision (%)	TOP20 Precision (%)	TOP50 Precision (%)	TOP100 Precision (%)	TOP200 Precision (%)
Lin functions	BiGRU	98.0	**99.0**	**96.0**	89.1	83.6
	BiLSTM	**99.1**	95.2	92.2	91.1	85.0
	TextCNN	97.0	96.2	94.1	**92.1**	**86.6**
Lin files	BiGRU	81.8	85.7	64.7	50.5	41.8
	BiLSTM	72.7	66.7	66.7	59.2	59.2
	TextCNN	81.8	71.4	54.9	39.6	28.9

The bold value indicates the best performance in the same performance metric of different methods

patterns are expressed in a language that is unique to the area. It acts as a unified intermediate program representation for all of Joern's supported languages. Then, a similar study that uses an attention mechanism was proposed in [89].

After that, DP-Transformer with improved learning ability was applied in software defect prediction in [90]. Their transformer network consists of stacked self-attention and position-wise, fully connected layers for both encoder and decoder. After each input sequence was inputted to the encoder, the decoder would generate a symbol output of an element. Especially, only the encoder part of the transformer was used to extract features from the source code.

After that, the Bert model was also applied to SVD in paper [91]. The authors adopted a neural network including 12 transformer blocks and a softmax layer. Their training includes two stages: pre-training on English Wikipedia data set [94] and fine-tuning process on vulnerability detection tasks. In the fine-tuning process, the SARD data set was used for vulnerability detection tasks.

The reviewed studies proved that the CNN and Recurrent Neural Network (RNN) were able to learn high-level representations for software defect prediction. Moreover, they have shown that graph-based code representations, such as AST and CFG, could represent semantic and syntactic information. However, the studies mentioned above convert the code representations to sequences before feeding them to the deep network instead of processing their original tree/graph form. The study proposed in [14] changed the state. The authors combined AST, CFG, DFG, and code sequence into a joint graph to comprehensively represent the semantic and syntactic information. Then, the gated graph recurrent layers [95] were adopted to learn the input graph structure. The main idea of this method was to combine multiple code representation methods to obtain more dense local features. Then, another GNN based method was proposed in [33]. The authors believed that structured information could better retain vulnerability features. Moreover, they applied a graph-based neural network to capture the graph representations of code explicitly. In this paper, the ASTs and control-data flow graphs (CDFGs) were used for code representations. Later, Cao et al. [93] conducted a Bidirectional Graph Neural-Network (BGNN) to improve the performance of DL-based vulnerability detection approaches.

(2)*Discussion*: In this section, some researchers strengthen CNN or other networks by introducing new mechanisms. For example, Lin et al. [68] adopted a global max-pooling layer to capture the most important signals; Duan et al. [88] combined an attention mechanism with CNN to capture the critical features.

Some researchers have used advanced neural networks for vulnerability detection, such as Transformer [90] and

Table 11 Test results of real-world data set Lin with different code representations

Data set	FPR(%)	TPR(%)	P(%)	F1(%)
ASTs	**1.1**	75.2	64.8	69.6
CDGs	1.3	76.5	61.0	67.9
CFGs	1.3	78.8	59.8	68.0
CPGs	2.1	85.4	53.5	65.8
PDGs	1.7	80.8	53.7	64.5
Source codes	**1.1**	**87.1**	66.2	75.3
Combined all	1.6	86.2	**68.9**	**76.5**

The bold value indicates the best performance in the same performance metric of different methods

Table 12 The number of vulnerable and non-vulnerable functions on different code representations

Feature	of vul. functions	of total functions
ASTs	1329	51,418
CDGs	1329	51,395
CFGs	1329	51,408
CPGs	1329	51,347
PDGs	1329	51,368
Source Codes	1471	60,768

Table 13 Test results of methods, trained and tested based on real-world data set Lin

System	FPR(%)	TPR(%)	P(%)	F1(%)
Flawfinder	12.3	28.9	5.4	9.1
Cppcheck	2.0	7.4	8.8	8.0
DNN	**0.6**	81.6	**75.8**	**78.6**
GRU	0.8	83.3	71.6	77.0
LSTM	1.4	84.3	59.4	69.7
BiGRU	1.0	85.8	69.0	76.4
BiLSTM	1.1	87.1	66.2	75.3
BiLSTM with Attention	1.1	86.8	65.4	74.6
TextCNN	2.4	**89.9**	48.2	62.8
CodeBERT	1.2	67.8	27.0	38.6
CodeBERT with BiLSTM	2.0	86.2	55.2	67.3

The bold value indicates the best performance in the same performance metric of different methods

Table 14 Cross-domain test results of methods, trained based on real-world data set Lin, tested based on real-world data set REVEAL

System	FPR(%)	TPR(%)	P(%)	F1(%)
Flawfinder	11.8	26.2	4.9	8.3
Cppcheck	2.3	7.6	8.9	8.2
DNN	1.9	5.5	24.0	9.0
GRU	2.9	10.2	27.6	15.0
LSTM	5.8	15.1	22.4	18.0
BiGRU	1.0	9.6	25.7	14.0
BiLSTM	4.8	14.8	25.0	18.6
BiLSTM with Attention	4.7	12.7	22.8	16.3
TextCNN	2.4	18.3	22.1	20.0
CodeBERT	1.6	28.1	26.7	27.4
CodeBERT with BiLSTM	**3.3**	**31.0**	**37.5**	**34.1**

The bold value indicates the best performance in the same performance metric of different methods

Bert [91]. It is worth noting that more and more researchers have applied GNN to SVD, such as [14, 33, 93]. Compared with other neural networks, GNN can retain more structural features [4].

3.5 Optimization for specific scenarios

This section shows some essential works that focus on optimizing specific scenarios in DL-based SVD.

(1) *Summary of recent works*: The cold-start problem is common in DL-based methods, and it means that ML tools usually can not play a good role due to the lack of high-quality data sets. Liu et al. proposed a method to break the dilemma of insufficient data sets in [96]. The author hoped to learn the common vulnerability features in the same type of data sets to perform the test set without labels better, and They adopted a metric transfer learning framework (MTLF). In MTLF, the target domain's Mahalanobis distance metric is computed by maximizing within-class covariance and minimizing between-class covariance. It could avoid the influence caused by the distribution difference between the target domain and the source domain.

A different method for improving the efficiency of cross-domain detection was proposed in [97]. First, the authors tested to bridge the distribution divergence between source and target projects by combining adversarial learning with discriminative feature learning, extracting the transferable semantic features from source code. In order to achieve this goal, they trained an Adversarial Discriminative Convolutional Neural Network (ADCNN) model. There were two independent training stages. The labeled source data was used to train the source encoder and source classifier in the first stage. Moreover, in the second stage, the authors trained the target encoder to make the target data representation similar to the source data representation by fooling the discriminator. Finally, the authors fed the features generated into a Logistic Regression (LR) classifier. The experimental results demonstrated that the proposed method performs better compared with other related cross-project defect prediction methods. Later, a extended method of [97] was proposed in [98]. The authors pointed out that the method proposed in [97] has negative impacts on the predictive performance due to the mode collapsing problem of the GAN principle. To tackle this problem, the authors adopted a Dual Generator-Discriminator Deep Code Domain Adaptation Network (Dual-GD-DDAN).

It is also very important to apply the detection system to an Integrated Development Environment (IDE) environment. In this way, the vulnerability can be corrected as soon as possible. A tool integrated with IDE as a plugin was developed in [99]. This tool worked in the background and could label vulnerability codes in the IDE environment. In this tool, ASTs created from the source code were used as the deep representation, and a three-layer neural network was used as a classifier. Therefore, this tool could detect code vulnerabilities in real-time during software development. The experiments were conducted on both open-source codes and Cisco codebases for C and C++ programming languages. The results showed that the method was an assuring approach for predicting vulnerabilities.

An interpretable model is also urgently needed. The DL models are considered black boxes because of the difficulty in explaining the relationship between input and output.

Table 15 Test results of real-world data set REVEAL in slice-level

System	FPR(%)	TPR(%)	P(%)	F1(%)
BiLSTM	11.0	13.9	17.7	15.6
TextCNN	**9.9**	10.9	20.6	14.3
GNN	10.8	**26.7**	**33.5**	**29.7**

The bold value indicates the best performance in the same performance metric of different methods

Therefore, it is not conducive to the understanding and confirmation of the output results. A method to determine the influence of local input on output was proposed in [100]. The authors combined two techniques to realize this method: 1) Syntax-Directed Attention; 2) Code Perturbation. Specifically, they used the attention mechanism score as the standard to determine the impact of the disturbance code on the output results by observing the change in the attention mechanism score. The experiments on more than 1000 programs indicated that attention scores could explain the output of DL-based SVD models.

(2)*Discussion*: This section mainly introduced the works which solve some specific problems in DL-based SVD, such as cold start [36], cross-project detection [96], and the interpretability of neural networks [100]. Among them, cold start and cross-project vulnerability detection are very real problems. Because in the actual detection, the software to be detected lacks labeled vulnerability data. It will seriously limit the vulnerability effect of software vulnerability detection. The interpretability of DL models will affect the confirmation and repair of software vulnerabilities. The current research methods can alleviate these problems to a certain extent, but they still face many limitations in the virtual environment. The research on vulnerability detection based on DL is still in its infancy. We hope that more research can explore and solve these practical problems.

Table 16 Test results of BiLSTM trained based on real-world data set Lin with different embedding methods

Embedding	FPR(%)	TPR(%)	P(%)	F1(%)
word2vec	1.1	**87.1**	66.2	75.3
fasttext	**0.7**	84.0	**73.1**	**78.2**
GloVe	2.5	22.9	18.0	20.2
doc2vec	2.2	18.8	17.0	17.9
CodeBERT	2.0	86.2	55.2	67.3

The bold value indicates the best performance in the same performance metric of different methods

Table 17 Cross-domain test results of BiLSTM, trained based on real-world data set Lin with different embedding methods, tested based on real-world data set REVEAL

Embedding	FPR(%)	TPR(%)	P(%)	F1(%)
word2vec	4.8	14.8	25.0	18.6
fasttext	7.8	16.5	18.8	17.6
GloVe	3.5	13.8	30.1	18.9
doc2vec	4.4	13.9	25.7	18.0
CodeBERT	**3.3**	**31.0**	**37.5**	**34.1**

The bold value indicates the best performance in the same performance metric of different methods

4 Challenges and future directions

This section conducts experiments based on real-data sets and draws conclusive remarks on research challenges and future trends based on previous works. The computational system used was a server running Ubuntu LTS 22.04 with two Physical Intel(R) Xeon(R) E5-2683 v4 2.00GHz CPUs and 32GB RAM with NVIDIA RTX 3090 GPUs. The main models involved in experiments were from GitHub.[8]

4.1 The lack of large-scale real-world benchmark data sets

The DL-based SVD methods need training on large-scale real-world data sets to achieve optimal performance [4]. At present, the lack of large-scale data sets containing high-quality vulnerability labels limits the research progress in this field. Table 7 summarizes the characteristics of a few popular software vulnerability data sets used by reviewed works.

It can be seen from Table 7 that more than half of the articles use synthetic or semi-synthetic data sets. The SARD[9] data set is the most widely used data set because this data set is open and easy to obtain and has a relatively large scale. However, this data set was initially designed for evaluating traditional vulnerability prediction based on static and dynamic analysis [102]. Therefore, the source codes of this data set are simplified and isolated. Research [57] compared The SARD data set with the real-world data set they collected. They find that the SARD data set and real-world data sets differ significantly in code complexity measurement. The main drawback is that the code patterns lack diversity compared to the code from real-world programs.

In addition, other synthetic or semi-synthetic data sets also have similar problems, such as CJOC-bAbI [48] and

[8] https://github.com/DanielLin1986/Function-level-Vulnerability-Detection.

[9] https://samate.nist.gov/SARD/testsuite.php.

Table 18 Cross-domain test results of different methods, trained based on real-world data set Lin, tested based on real-world data set REVEAL

System	FPR(%)	TPR(%)	P(%)	F1(%)
BiLSTM	4.8	14.8	25.0	18.6
AD-CNN [97]	3.5	40.6	28.3	33.4
CD-VuID [96]	3.2	37.6	35.8	36.7

s-bAbI [63]. Besides, the scale of these two data sets is relatively small, and the source codes of the data set CJOC-bAbI cannot even be compiled. Furthermore, the data set PROMISE[10] cannot be found on the website provided in the study [32] due to the lack of maintenance for a long time. We extract the data set from the website mirror on GitHub.[11] This data set only provides the static features extracted from the software source codes from the obtained data. Therefore, it is not conducive to furthermore research based on this data set.

The synthetic or semi-synthetic data sets do not fully capture the complexities of real-world vulnerabilities [102–104]. Many existing works created self-constructed data sets based on different criteria. However, only a few fully released their data sets. Lin et al. released the data set used in their study [66]. Due to the manual extraction of vulnerability source codes based on CVE[12] information, the real-world data set provided by Lin et al. has been used in many studies, such as [67, 98]. In addition, high-quality labels make this data set have the potential to become a small-size benchmark data set at function-level.

However, due to labor costs, the scale of Lin is small. Table 8 lists the test results of the detection ability based on the real data set Lin, and the semi-synthetic data set SARD. The details of the data sets are shown in Table 9, there is no duplication between the training set and the test set (The bold value indicates the best performance in the same performance metric of different methods). It can be seen from the experimental results that the detection results based on real data sets are significantly better than semi-synthetic data sets. However, it still with high false positive rates due to insufficient data scale. For more general cases, large-scale real-world benchmark data sets are still needed. Such data sets could facilitate all research works in this field, and the comparative experiments carried out on the data set could fairly reveal the differences between different works. Of course, the granularity and label quality of the data sets are as important as the data scale. It can be seen from table 10 that the deep learning models trained based on the file-level data set performed better than the models trained based on the function-level data set.

However, other real-world data sets, such as Draper [102], D2A [70], have a large scale, but the quality of the labels needs to be verified. For example, the labels of the data set Draper are mainly from static detection tools. And the labels of data set D2A are from the analysis of software source codes in different versions. Although their labeling method is reasonable, there is still a big gap between their methods with labeling based on CVE information.

Indeed, there may be potential risks in open source vulnerability data sets, but a large-scale vulnerability data set with a high-quality label is essential. Federated Learning (FL) may alleviate this problem for some data sets that are not suitable to release because this method can share features without sharing data sets [105].

4.2 Effective code representations

In order to optimize the performance of DL-based SVD, researchers have proposed a variety of code representation methods to provide neural networks with richer semantic and syntactic features. Furthermore, it has been proven that code representation methods can preserve more useful structure information of source code, resulting in the best performance balance between precision and recall [102, 106].

The current researchers attempt to integrate various code representation methods to contain more information to extract vulnerability features better. For example, some studies extract features from source code-related information such as code comments [75] and binary files [52]. These methods play a positive role in improving detection performance.

In addition, many researchers use code structure analysis to optimize code representation methods, such as AST [68]. A tree representation of the abstract syntactic structure of a source code written in a formal language is known as an AST. Each node of the tree indicates a construct that appears in the text. Moreover, they add source codes' data flow, control flow, and other structural information to the AST structure, forming various code representation methods such as CPG [88], CDFG [33], code gadgets [92]. All possible routes through a program during execution are referred to as a program's control flow. And the data flow monitors the control flow's use of variables.

Table 12 lists the number of different code representations generated by Joern based on the same data set. The number of generated code representations is not the same as the number of source code samples. It is because the code representations of some source codes are empty. Table 11 shows that the detection method trained based on

[10] http://openscience.us/repo/defect.

[11] https://github.com/opensciences/opensciences.github.io.

[12] https://cve.mitre.org/

source code has even better performance than that based on the single code representation methods. Although the combined code representations perform better than single methods, the scale of the code representation to input neural networks is also limited due to hardware limitations. Thus, while adding these pieces of information to AST, researchers also need to constantly remove the features of low values, forming a dense feature representation.

How to extract the most important features from combined representations to input them into the neural networks full is research worthy of attention. However, detecting different vulnerabilities needs to retain features due to the diversity of vulnerability patterns. Therefore, customizing appropriate code representation methods for specific types of vulnerabilities may contribute to the detection performance of specific vulnerabilities. For example, at present, there are code representation methods suitable for buffer overflow vulnerability detection [63]. However, for most other types of vulnerabilities, there are no specialized code representation methods. Therefore, designing the optimal code representation methods for specific programming languages and vulnerability types may be an important research topic in future.

4.3 Humanoid DL model

Researchers' goal is that the neural model can detect the vulnerable code like human experts do and pinpoint the relevant code leading to the vulnerability. In order to achieve this goal, more complex models with stronger learning abilities have been applied to this field, from simple such as DBN [32], MLP [52] to relatively complex such as CNN [49], LSTM [22]. It can be seen from tables 13 and 14 that even the DNN model has better performance than the traditional detection tools: Flawfinder and Cppcheck. However, DL-based SVD methods have poor stability in cross-domain detection due to the difference in data distribution.

Besides, these models cannot focus on and fully learn important features or pinpoint the critical code lines that affect the output results. Therefore, the attention mechanism with this ability has been sought after by researchers. For example, research [88] and [89] applied the attention mechanism on CNN and RNN networks and proved that the attention mechanism is indeed helpful in improving the performance of DL-based SVD. In addition, study [37] and [100] used the attention mechanism to calculate the weight of different code lines, which showed that the attention mechanism helps pinpoint the critical code lines that lead to vulnerabilities.

In addition, Transformer [90] and Bert [91], which are completely composed of attention mechanisms, have also been applied in this field. As a result, these models have stronger learning abilities than the previous models. In research [91], the authors trained the BERT model in English writing data set and then used this model to detect vulnerabilities.

We believe that humanoid DL models maybe appear with the continuous development of neural networks, which can help human experts complete most of the tedious vulnerability detection work. At present, many advanced deep learning models such as meta-learning [107] have been proposed. Moreover, some of them are especially suitable for software code analysis, such as CodeBERT [83]. Due to the limitation of the hardware platform, we did not train the CodeBERT model locally but used the embedding model trained by the research [83] to fine-tune it. We hope that research can fully tap the potential of this model in the field of SVD. As these models are applied to the field of SVD, the SVD methods can further improve the detection accuracy.

4.4 Semantic retention in neural networks

Notably, the neural models may not be able to capture the code semantics as human experts do.

On the one hand, the loss of semantic information is inevitably in the process of model training. However, the semantic retention of each neural network model is different. For example, previous studies have shown that the CNN network has advantages in local feature extraction [49, 80], and the BiLSTM network is better in long sequences' feature extraction [24, 69]. However, in the field of vulnerability detection, people's understanding of the preference of different neural networks for semantic retention is not clear [76]. How to select a suitable neural network to preserve relevant semantic features of vulnerabilities needs further research.

On the other hand, semantic loss occurs more before the code representations are input to neural networks. Due to hardware limitations, the code representations are usually limited to a fixed length in the input process. It means that the code representations which exceed the size will be truncated [68]. In real-world data sets, there are a lot of lengthy codes. When using these codes for training or detection, there is a severe problem of semantic loss. However, this review has not found the research dealing with this problem.

At the same time, many code representation methods, such as AST and CFG, are graph structures. However, for most neural network models, such as RNN, the graph structure needs to be transformed into a sequence to input. As a result, the structural features may be lost in the process. GNN model can directly input structured information, which has attracted the attention of researchers [14, 33, 93]. From the experimental results in table 15, it

can be seen that the GNN network does have a better learning ability at the slice-level granularity. However, due to the input limitations of the GNN, the code needs to be sliced and processed into a graph structure. High-quality labels for slice-level data sets are difficult to meet. The performance of the test results of GNN trained by slice-level data set is not as good as other neural networks trained by function-level data sets. SVD based on GNNs still needs further research.

Besides, embedding methods also deserve to be attention. Table 16 shows the test results of the same BiLSTM model trained based on different embedding methods. The performance of GloVe and Doc2vec models perform poorly. It is because GloVe focuses on word co-occurrence, and Doc2vec hopes to extract sentence vectors and article vectors. However, compared with natural languages, the semantic information of programming languages is mainly contained in naming functions and variables. So it is not easy to learn semantic features for GloVe and Doc2vec. Similarly, the performance of the CodeBERT method is also reduced. However, because the CodeBERT model is pre-trained on large-scale code, it learned richer semantic information than GloVe and Doc2vec. In the cross-domain detection, as shown in Table 17, CodeBERT achieved the best performance.

On the whole, how to better retain the semantic information of code needs further research.

4.5 Vulnerability detection in the cross-environment

Another problem worthy of attention is that the current DL-based vulnerability detection methods have a narrow range of applicability.

One is cross programming language environment. For example, most studies are limited to detecting part of vulnerabilities (such as buffer overflow) written in several mainstream programming languages (such as C, Java). Due to software source codes usually involving multiple programming languages and vulnerabilities, the detection ability of these methods can not meet the detection requirements. Zou et al. [19] tried to develop a multi-class vulnerability detection system to cover most classes of vulnerabilities. However, limited by the small number of data samples, the actual detection effect of these vulnerabilities is still not guaranteed. Besides, Li et al. [78] developed a detection system that can detect vulnerabilities in different languages at the same time. However, the model's design is still based on these mainstream languages, which can not guarantee the same detection efficiency in other languages. Therefore, it is worthy of further study on the DL-base SVD for multiple programming languages and multiple types of vulnerabilities.

The other is cross-project environments. Due to the differences in dependent function libraries, DL-based SVD methods often have low accuracy in cross-project detection [68]. However, cross-project detection is inevitable in the actual detection environment because the software code to be detected usually has no label [96]. In order to alleviate this problem, researchers have proposed some cross-project detection methods. For example, Liu et al. [96] learned the cross-project representation by minimizing the distribution difference between the source and target domains to improve cross-project detection efficiency. Furthermore, Sheng et al. [97] tested to bridge the distribution divergence between source and target projects by combining adversarial learning with discriminative feature learning. However, as shown in table 18, although these studies have improved cross-project vulnerability detection performance, the reduced accuracy problem can not be completely avoided. Therefore, more research on cross-project vulnerability detection is needed to better meet the actual detection needs.

5 Conclusion

With the advancement of artificial intelligence technology, a software vulnerability detection system based on deep learning may achieve autonomous and intelligent vulnerability mining, successfully avoiding the issues of large false-positive and false-negative vulnerability rates. This research comprehensively evaluates available deep learning-based software vulnerability detection algorithms and examines the perception gap. At the same time, it covers the current state of research and trends in software vulnerability detection approaches based on deep learning methods to address this issue. Finally, this field's difficulties and prospects have been identified.

We believe that the two most important problems are to be solved in this field. The first is the lack of large-scale public data sets, making it difficult for various methods in the current field to compare under objective conditions. The second is the cross-environment vulnerability detection method because only by ensuring the detection result in the real environment can this method be applied to the real world. We hope that more research will focus on solving these problems in future.

Acknowledgements This work was supported by the National Natural Science Foundation of China [grant number 61772478]. (Yuhui zhu and Guanjun Lin contributed equally to this work.)

Data availability The datasets generated during and analyzed during the current study are available from the corresponding author upon reasonable request.

Declarations

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

References

1. Liu L, De Vel O, Han QL, Zhang J, Xiang Y (2018) Detecting and preventing cyber insider threats: a survey. IEEE Commun Surveys Tutorials 20(2):1397–1417
2. Wang M, Zhu T, Zhang T, Zhang J, Yu S, Zhou W (2020) Security and privacy in 6g networks: new areas and new challenges. Digital Commun Netw 6(3):281–291
3. Techniques NR, Expose HDD, Target A, Lucrative M (2019) Mcafee labs threats report: December 2018. Comput Fraud Secur 2019(1):4. https://doi.org/10.1016/S1361-3723(19)30004-1
4. Lin G, Wen S, Han QL, Zhang J, Xiang Y (2020) Software vulnerability detection using deep neural networks: a survey. In: Proceedings of the IEEE pp 1–24, https://doi.org/10.1109/JPROC.2020.2993293
5. Ghaffarian SM, Shahriari HR (2017) Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey. Acm Comput Surv 50(4):56. https://doi.org/10.1145/3092566
6. Engler D, Chen D, Hallem S, Chou A, Chelf B (2001) Bugs as deviant behavior: a general approach to inferring errors in systems code. Symposium on operating systems principles 35. https://doi.org/10.1145/502034.502041
7. Liang H, Wang L, Wu D, Xu J (2016) MLSA: a static bugs analysis tool based on LLVM IR. Int J Netw Distrib Comput 4:137. https://doi.org/10.2991/ijndc.2016.4.3.1
8. Cassez F, Sloane AM, Roberts M, Pigram M, Suvanpong P, de Aledo Marugán PG (2017) Skink: Static analysis of programs in LLVM intermediate representation - (competition contribution). In: Legay A, Margaria T (eds) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II, Lecture Notes in Computer Science, vol 10206, pp 380–384, https://doi.org/10.1007/978-3-662-54580-5_27,
9. Jang J, Agrawal A, Brumley D (2012) ReDeBug: finding unpatched code clones in entire os distributions. In: IEEE pp 48–62, https://doi.org/10.1109/SP.2012.13
10. Li H, Kwon H, Kwon J, Lee H (2014) A scalable approach for vulnerability discovery based on security patches. In: Batten L, Li G, Niu W, Warren M (eds) Applications and techniques in information security. Springer, Berlin, Heidelberg, pp 109–122
11. Scandariato R, Walden J, Hovsepyan A, Joosen W (2014) Predicting vulnerable software components via text mining. IEEE Trans Softw Eng 40:993–1006. https://doi.org/10.1109/TSE.2014.2340398
12. Wang Y, Jia P, Liu L, Liu J (2019) A systematic review of fuzzing based on machine learning techniques. PloS one 15(8):e0237749
13. Yamaguchi F, Golde N, Arp D, Rieck K (2014) Modeling and discovering vulnerabilities with code property graphs. In: proceedings - IEEE symposium on security and privacy, https://doi.org/10.1109/SP.2014.44
14. Zhou Y, Liu S, Siow J, Du X, Liu Y (2019) Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv Neural Inf Process Syst, pp 10197–10207
15. Yamaguchi F, Wressnegger C, Gascon H, Rieck K (2013) Chucky: exposing missing checks in source code for vulnerability discovery. In: proceedings of the acm conference on computer and communications security, https://doi.org/10.1145/2508859.2516665
16. Yamaguchi F, Maier A, Gascon H, Rieck K (2015) Automatic inference of search patterns for taint-style vulnerabilities. vol 2015, https://doi.org/10.1109/SP.2015.54
17. Shankar U, Talwar K, Foster J, Wagner D (2001) Detecting format string vulnerabilities with type qualifiers. USENIX Security 10
18. Shin Y, Meneely A, Williams L, Osborne J (2011) Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans Softw Eng 37:772–787. https://doi.org/10.1109/TSE.2010.81
19. Zou D, Wang S, Xu S, Li Z, Jin H (2019) μVulDeePecker: A deep learning-based system for multiclass vulnerability detection. IEEE Trans Depend Secure Comput
20. Sun N, Zhang J, Rimba P, Gao S, Zhang LY, Xiang Y (2019) Data-driven cybersecurity incident prediction: a survey. IEEE Commun Surveys Tutorials 21(2):1744–1772
21. Miao Y, Chen C, Pan L, Han QL, Zhang J, Xiang Y (2021) Machine learning based cyber attacks targeting on controlled information: a survey. ACM Comput Survey 54(7):1–36
22. Lin G, Zhang J, Luo W, Pan L, Xiang Y (2017) POSTER: vulnerability discovery with function representation learning from unlabeled projects. In: Thuraisingham BM, Evans D, Malkin T, Xu D (eds) Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, ACM, pp 2539–2541, https://doi.org/10.1145/3133956.3138840
23. Chen X, Li C, Wang D, Wen S, Zhang J, Nepal S, Xiang Y, Ren K (2020) Android hiv: a study of repackaging malware for evading machine-learning detection. IEEE Trans Inf Forensics and Secur 15:987–1001
24. Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018) VulDeePecker: a deep learning-based system for vulnerability detection. In: 25th annual network and distributed system security symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018, The Internet Society
25. Qiu J, Zhang J, Pan L, Luo W, Nepal S, Xiang Y (2020) A survey of android malware detection with deep neural models. ACM Computi Survey 53(6):126:1-126:31
26. Liu B, Shi L, Cai Z, Li M (2012) Software vulnerability discovery techniques: a survey. In: 2012 fourth international conference on multimedia information networking and security, pp 152–156, https://doi.org/10.1109/MINES.2012.202
27. Malhotra R (2015) A systematic review of machine learning techniques for software fault prediction. Appl Soft Comput 27:504–518. https://doi.org/10.1016/j.asoc.2014.11.023
28. Ji T, Wu Y, Wang C, Zhang X, Wang Z (2018) The coming era of alphahacking?: a survey of automatic software vulnerability detection, exploitation and patching techniques. In: 2018 IEEE third international conference on data science in cyberspace (DSC), pp 53–60, https://doi.org/10.1109/DSC.2018.00017
29. Allamanis M, Barr ET, Devanbu PT, Sutton C (2018) A survey of machine learning for big code and naturalness. ACM Comput Surv 51(4):81:1-81:37. https://doi.org/10.1145/3212695
30. Shahriar H, Zulkernine M (2012) Mitigating program security vulnerabilities. ACM Comput Surveys 44(3):1–46. https://doi.org/10.1145/2187671.2187673
31. Jie G, Xiao-Hui K, Qiang L (2016) Survey on software vulnerability analysis method based on machine learning. In: 2016

IEEE first international conference on data science in cyberspace (DSC), pp 642–647, https://doi.org/10.1109/DSC.2016.33
32. Wang S, Liu T, Tan L (2016) Automatically learning semantic features for defect prediction. In: 2016 IEEE/ACM 38th international conference on software engineering (ICSE), pp 297–308, https://doi.org/10.1145/2884781.2884804
33. Brauckmann A, Goens A, Ertel S, Castrillón J (2020) Compiler-based graph representations for deep learning models of code. In: Pouchet L, Jimborean A (eds) CC '20: 29th International Conference on Compiler Construction, San Diego, CA, USA, February 22-23, 2020, ACM, pp 201–211, https://doi.org/10.1145/3377555.3377894,
34. Viet Phan A, Le Nguyen M, Thu Bui L (2017) Convolutional neural networks over control flow graphs for software defect prediction. In: 2017 IEEE 29th international conference on tools with artificial intelligence (ICTAI), IEEE, Boston, MA, pp 45–52, https://doi.org/10.1109/ICTAI.2017.00019
35. Xiaomeng W, Tao Z, Runpu W, Wei X, Changyu H (2018) CPGVA: code property graph based vulnerability analysis by deep learning. In: 2018 10th international conference on advanced infocomm technology (ICAIT), IEEE, pp 184–188
36. Lin G, Zhang J, Luo W, Pan L, De Vel O, Montague P, Xiang Y (2019) Software vulnerability discovery via learning multi-domain knowledge bases. IEEE Trans Depend Secure Comput 18(5):2469–2485. https://doi.org/10.1109/TDSC.2019.2954088
37. Srikant S, Lesimple N, O'Reilly UM (2020) Dependency-Based Neural Representations for Classifying Lines of Programs. arXiv preprint arXiv:2004.10166 2004.10166
38. Nirmal I, Khamis A, Hassan M, Hu W, Zhu X (2021) Deep learning for radio-based human sensing: recent advances and future directions. IEEE Commun Surv Tutorials 23(2):995–1019. https://doi.org/10.1109/COMST.2021.3058333
39. Feriani A, Hossain E (2021) Single and multi-agent deep reinforcement learning for AI-enabled wireless networks: a tutorial. IEEE Commun Surv Tutorials 23(2):1226–1252. https://doi.org/10.1109/COMST.2021.3063822
40. Chen W, Qiu X, Cai T, Dai H, Zheng Z, Zhang Y (2021) Deep reinforcement learning for internet of things: a comprehensive survey. IEEE Commun Surv Tutorials 23(3):1659–1692. https://doi.org/10.1109/COMST.2021.3073036
41. Romero J, Machado P (2021) Neural networks in art, sound and design. Neural Comput Appl 33(1):1. https://doi.org/10.1007/s00521-020-05444-y
42. Briot J (2021) From artificial neural networks to deep learning for music generation: history, concepts and trends. Neural Comput Appl 33(1):39–65. https://doi.org/10.1007/s00521-020-05399-0
43. Chitradevi D, Prabha S, Prabhu AD (2021) Diagnosis of alzheimer disease in MR brain images using optimization techniques. Neural Comput Appl 33(1):223–237. https://doi.org/10.1007/s00521-020-04984-7
44. Bhandari AK, Rahul K, Shahnawazuddin S (2021) A fused contextual color image thresholding using cuttlefish algorithm. Neural Comput Appl 33(1):271–299. https://doi.org/10.1007/s00521-020-05013-3
45. Singh M, Kumar R, Chana I (2021) Improving neural machine translation for low-resource indian languages using rule-based feature extraction. Neural Comput Appl 33(4):1103–1122. https://doi.org/10.1007/s00521-020-04990-9
46. Sitender BS (2021) A sanskrit-to-english machine translation using hybridization of direct and rule-based approach. Neural Comput Appl 33(7):2819–2838. https://doi.org/10.1007/s00521-020-05156-3
47. Mohan K, Seal A, Krejcar O, Yazidi A (2021) Fer-net: facial expression recognition using deep neural net. Neural Comput Appl 33(15):9125–9136. https://doi.org/10.1007/s00521-020-05676-y
48. Choo J, Choi Mj, Jeong S, Oh H (2017) End-to-End prediction of buffer overruns from raw source code via neural memory networks pp 1546–1553
49. Li J, He P, Zhu J, Lyu MR (2017) Software Defect Prediction via Convolutional Neural Network. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, Prague, Czech Republic, pp 318–328, https://doi.org/10.1109/QRS.2017.42
50. Pradel M, Sen K (2017) Deep learning to find bugs. TU Darmstadt Dep Comput Sci, 4(1)
51. Wu F, Wang J, Liu J, Wang W (2017) Vulnerability detection with deep learning. In: 2017 3rd IEEE international conference on computer and communications (ICCC), IEEE, Chengdu, pp 1298–1302, https://doi.org/10.1109/CompComm.2017.8322752
52. Grieco G, Grinblat GL, Uzal L, Rawat S, Feist J, Mounier L (2016) Toward large-scale vulnerability discovery using machine learning. In: proceedings of the sixth ACM conference on data and application security and privacy, association for computing machinery, New York, NY, USA, CODASPY '16, pp 85–96, https://doi.org/10.1145/2857705.2857720
53. Kim S, Zhang H, Wu R, Gong L (2011) Dealing with noise in defect prediction. In: 2011 33rd international conference on software engineering (ICSE), pp 481–490, https://doi.org/10.1145/1985793.1985859
54. Witten Ian H (2011) EF (2011). Data Mining: Practi Mach Learn Tools Tech 31:6. https://doi.org/10.1016/C2009-0-19715-5
55. Jureczko M, Madeyski L (2010) Towards identifying software project clusters with regard to defect prediction. In: ACM international conference proceeding series, vol 9, p 9, https://doi.org/10.1145/1868328.1868342
56. He Z, Peters F, Menzies T, Yang Y (2013) Learning from open-source projects: an empirical study on defect prediction. In: international symposium on empirical software engineering and measurement, pp 45–54, https://doi.org/10.1109/ESEM.2013.20
57. Chakraborty S, Krishna R, Ding Y, Ray B (2020) Deep learning based vulnerability detection: Are we there yet? CoRR abs/2009.07235, https://arxiv.org/abs/2009.07235, 2009.07235
58. Zhang J, Pan L, Han QL, Chen C, Wen S, Xiang Y (2021) Deep learning based attack detection for cyber-physical system cybersecurity: a survey. IEEE/CAA J Autom Sinica. https://doi.org/10.1109/JAS.2021.1004261
59. Mikolov T, Sutskever I, Chen K, Corrado G, Dean J (2013b) Distributed representations of words and phrases and their compositionality. Adv Neural Inf Process Syst, 26
60. Mikolov T, Chen K, Corrado G, Dean J (2013a) Efficient estimation of word representations in vector space. arXiv:1301.3781 [cs] 1301.3781
61. Fan G, Diao X, Yu H, Yang K, Chen L (2019a) Deep Semantic Feature Learning with Embedded Static Metrics for Software Defect Prediction. In: 2019 26th Asia-Pacific Software Engineering Conference (APSEC), IEEE, Putrajaya, Malaysia, pp 244–251, https://doi.org/10.1109/APSEC48747.2019.00041
62. Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M (2018) Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE international conference on machine learning and applications (ICMLA), pp 757–762, https://doi.org/10.1109/ICMLA.2018.00120
63. Sestili CD, Snavely WS, VanHoudnos NM (2018) Towards security defect prediction with AI. arXiv:1808.09897 [cs, stat] 1808.09897

64. Li Z, Zou D, Xu S, Chen Z, Zhu Y, Jin H (2020b) VulDeeLocator: a deep learning-based fine-grained vulnerability detector. CoRR abs/2001.02350
65. Liu S, Lin G, Han QL, Wen S, Zhang J, Xiang Y (2020) DeepBalance: deep-learning and fuzzy oversampling for vulnerability detection. IEEE Trans Fuzzy Syst 28(7):1329–1343. https://doi.org/10.1109/TFUZZ.2019.2958558
66. Lin G, Xiao W, Zhang J, Xiang Y (2019a) Deep learning-based vulnerable function detection: a benchmark. In: Zhou J, Luo X, Shen Q, Xu Z (eds) information and communications security - 21st international conference, ICICS 2019, Beijing, China, December 15-17, 2019, Revised Selected Papers, Springer, Lecture Notes in Computer Science, vol 11999, pp 219–232, https://doi.org/10.1007/978-3-030-41579-2_13,
67. Nguyen V, Le T, Le T, Nguyen K, DeVel O, Montague P, Qu L, Phung D (2019) Deep domain adaptation for vulnerable code function identification. In: 2019 international joint conference on neural networks (IJCNN), pp 1–8, https://doi.org/10.1109/IJCNN.2019.8851923
68. Lin G, Zhang J, Luo W, Pan L, Xiang Y, De Vel O, Montague P (2018) Cross-project transfer representation learning for vulnerable function discovery. IEEE Trans Ind Inf 14(7):3289–3297
69. Saccente N, Dehlinger J, Deng L, Chakraborty S, Xiong Y (2019) Project Achilles: a prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network. In: 2019 34th IEEE/ACM international conference on automated software engineering workshop (ASEW), pp 114–121, https://doi.org/10.1109/ASEW.2019.00040
70. Zheng Y, Pujar S, Lewis BL, Buratti L, Epstein EA, Yang B, Laredo J, Morari A, Su Z (2021b) D2A: A dataset built for AI-Based vulnerability detection methods using differential analysis. In: 43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021, IEEE, pp 111–120, 10/gkgd53, https://doi.org/10.1109/ICSE-SEIP52600.2021.00020
71. Wang H, Ye G, Tang Z, Tan SH, Huang S, Fang D, Feng Y, Bian L, Wang Z (2021b) Combining Graph-based learning with automated data collection for code vulnerability detection 16:1943–1958, 10/gkgf4k, https://ieeexplore.ieee.org/document/9293321/
72. Zhu X, Goldberg A (2009) Introduction to semi-supervised learning. Synth Lect Artif Intell Mach Learn 3(1):1–130
73. Grandvalet Y, Bengio Y (2004) Semi-supervised Learning by entropy minimization. Adv Neural Inform Process Syst, 17
74. Harer JA, Kim LY, Russell RL, Ozdemir O, Kosta LR, Rangamani A, Hamilton LH, Centeno GI, Key JR, Ellingwood PM, Antelman E, Mackay A, McConley MW, Opper JM, Chin P, Lazovich T (2018) Automated software vulnerability detection with machine learning. arXiv:1803.04497 [cs, stat] 1803.04497
75. Huo X, Yang Y, Li M, Zhan DC (2018) learning semantic features for software defect prediction by code comments embedding. In: 2018 IEEE international conference on data mining (ICDM), IEEE, Singapore, pp 1049–1054, https://doi.org/10.1109/ICDM.2018.00133
76. Li Z, Zou D, Tang J, Zhang Z, Sun M, Jin H (2019) A comparative study of deep learning-based vulnerability detection system. IEEE Access 7:103184–103197
77. Pradel M, Sen K (2018) DeepBugs: A learning approach to name-based bug detection. In: proceedings of the ACM on programming languages 2(OOPSLA):147:1–147:25, https://doi.org/10.1145/3276517
78. Li R, Feng C, Zhang X, Tang C (2019) A lightweight assisted vulnerability discovery method using deep neural networks. IEEE Access 7:80079–80092. https://doi.org/10.1109/ACCESS.2019.2923227
79. Zaharia S, Rebedea T, Trausan-Matu S (2019) Source code vulnerabilities detection using loosely coupled data and control flows. In: 2019 21st international symposium on symbolic and numeric algorithms for scientific computing (SYNASC), pp 43–46, https://doi.org/10.1109/SYNASC49474.2019.00016
80. Li X, Wang L, Xin Y, Yang Y, Chen Y (2020) Automated vulnerability detection in source code using minimum intermediate representation learning. Appl Sci 10(5):1692
81. Zheng W, Semasaba AOA, Wu X, Agyemang SA, Liu T, Ge Y (2021a) Representation vs. model: what matters most for source code vulnerability detection. In: 28th IEEE international conference on software analysis, evolution and reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021, IEEE, pp 647–653, 10/gk52qg, https://doi.org/10.1109/SANER50967.2021.00082
82. Tang G, Yang L, Ren S, Meng L, Yang F, Wang H (2021b) An automatic source code vulnerability detection approach based on KELM 2021:5566423:1–5566423:12, 10/gmbqfw, https://doi.org/10.1155/2021/5566423
83. Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020) Codebert: A pre-trained model for programming and natural languages. In: Cohn T, He Y, Liu Y (eds) Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020, Association for Computational Linguistics, Findings of ACL, vol EMNLP 2020, pp 1536–1547, https://doi.org/10.18653/v1/2020.findings-emnlp.139,
84. Lattner C, Adve V (2004) LLVM: A compilation framework for lifelong program analysis transformation. In: international symposium on code generation and optimization, 2004. CGO 2004., pp 75–86, https://doi.org/10.1109/CGO.2004.1281665
85. Le Q, Mikolov T (2014) Distributed representations of sentences and documents. In: proceedings of the 31st international conference on international conference on machine learning - volume 32, JMLR.org, ICML'14, pp II–1188–II–1196
86. Hin D, Kan A, Chen H, Babar MA (2022) Linevd: Statement-level vulnerability detection using graph neural networks. CoRR abs/2203.05181, https://doi.org/10.48550/arXiv.2203.05181, 2203.05181
87. Pennington J, Socher R, Manning CD (2014) Glove: global vectors for word representation. In: Moschitti A, Pang B, Daelemans W (eds) proceedings of the 2014 conference on empirical methods in natural language processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL, ACL, pp 1532–1543, https://doi.org/10.3115/v1/d14-1162,
88. Duan X, Wu J, Ji S, Rui Z, Luo T, Yang M, Wu Y (2019) VulSniper: Focus your attention to shoot fine-grained vulnerabilities. In: IJCAI, pp 4665–4671
89. Fan G, Diao X, Yu H, Yang K, Chen L (2019b) Software defect prediction via attention-based recurrent neural network. Scientific Programming 2019
90. Zhang Q, Wu B (2020) Software defect prediction via transformer. In: 2020 IEEE 4th information technology, networking, electronic and automation control conference (ITNEC), vol 1, pp 874–879, https://doi.org/10.1109/ITNEC48623.2020.9084745
91. Ziems N, Wu S (2021) Security vulnerability detection using deep learning natural language processing. In: 2021 IEEE Conference on Computer Communications Workshops, INFOCOM Workshops 2021, Vancouver, BC, Canada, May 10-13, 2021, IEEE, pp 1–6, 10.1109/INFOCOMWKSHPS51825.2021.9484500,

92. Tang G, Meng L, Ren S, Cao W, Wang Q, Yang L (2021a) A comparative study of neural network techniques for automatic software vulnerability detection abs/2104.14978, https://arxiv.org/abs/2104.14978, 2104.14978
93. Cao S, Sun X, Bo L, Wei Y, Li B (2021) BGNN4VD: constructing bidirectional graph neural-network for vulnerability detection. Inf Softw Technol 136:106576. https://doi.org/10.1016/j.infsof.2021.106576
94. Devlin J, Chang M, Lee K, Toutanova K (2019) BERT: pre-training of deep bidirectional transformers for language understanding. In: Burstein J, Doran C, Solorio T (eds) Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers), Association for Computational Linguistics, pp 4171–4186, https://doi.org/10.18653/v1/n19-1423,
95. Li Y, Tarlow D, Brockschmidt M, Zemel R (2015) Gated graph sequence neural networks
96. Liu S, Lin G, Qu L, Zhang J, De Vel O, Montague P, Xiang Y (2020b) CD-VulD: Cross-Domain Vulnerability Discovery based on Deep Domain Adaptation. IEEE Trans Dependable Secure Comput pp 1–1, https://doi.org/10.1109/TDSC.2020.2984505
97. Sheng L, Lu L, Lin J (2020) An adversarial discriminative convolutional neural network for cross-project defect prediction. IEEE Access 8:55241–55253. https://doi.org/10.1109/ACCESS.2020.2981869
98. Nguyen V, Le T, de Vel OY, Montague P, Grundy JC, Phung D (2020) Dual-component deep domain adaptation: A new approach for cross project software vulnerability detection. In: Lauw HW, Wong RC, Ntoulas A, Lim E, Ng S, Pan SJ (eds) Advances in Knowledge Discovery and Data Mining - 24th Pacific-Asia Conference, PAKDD 2020, Singapore, May 11-14, 2020, Proceedings, Part I, Springer, Lecture Notes in Computer Science, vol 12084, pp 699–711, https://doi.org/10.1007/978-3-030-47426-3_54,
99. Tanwar A, Sundaresan K, Ashwath P, Ganesan P, Chandrasekaran SK, Ravi S (2020) Predicting vulnerability in large codebases with deep code representation. arXiv preprint arXiv:2004.12783 2004.12783
100. Bui NDQ, Yu Y, Jiang L (2019) Autofocus: Interpreting attention-based neural networks by code perturbation. In: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, IEEE, pp 38–41, https://doi.org/10.1109/ASE.2019.00014,
101. Lin G, Xiao W, Zhang LY, Gao S, Tai Y, Zhang J (2021) Deep neural-based vulnerability discovery demystified: data, model and performance. Neural Comput Appl 33(20):13287–13300. https://doi.org/10.1007/s00521-021-05954-3
102. Liu Z, Qian P, Wang X, Zhu L, He Q, Ji S (2021) Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion. In: Zhou Z (ed) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021, ijcai.org, pp 2751–2759, https://doi.org/10.24963/ijcai.2021/379,
103. Ashizawa N, Yanai N, Cruz JP, Okamura S (2021) Eth2vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In: Gai K, Choo KR (eds) BSCI '21: Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure, Virtual Event, Hong Kong, June 7, 2021, ACM, pp 47–59, https://doi.org/10.1145/3457337.3457841,
104. Ding M, Li P, Li S, Zhang H (2021) Hfcontractfuzzer: Fuzzing hyperledger fabric smart contracts for vulnerability detection. In: Chitchyan R, Li J, Weber B, Yue T (eds) EASE 2021: Evaluation and Assessment in Software Engineering, Trondheim, Norway, June 21-24, 2021, ACM, pp 321–328, https://doi.org/10.1145/3463274.3463351,
105. Cao X, Jia J, Gong NZ (2021b) Provably secure federated learning against malicious clients. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021, AAAI Press, pp 6885–6893, https://ojs.aaai.org/index.php/AAAI/article/view/16849
106. Wu Y, Lu J, Zhang Y, Jin S (2021) Vulnerability detection in C/C++ source code with graph representation learning. In: 11th IEEE annual computing and communication workshop and conference, CCWC 2021, las vegas, NV, USA, january 27-30, 2021, IEEE, pp 1519–1524, 10/gmbqf6, https://doi.org/10.1109/CCWC51732.2021.9376145, tex.bibsource: dblp computer science bibliography, https://dblp.org tex.biburl: https://dblp.org/rec/conf/ccwc/WuLZJ21.bib tex.timestamp: Thu, 25 Mar 2021 08:31:10 +0100
107. Wang C, Qiu M, Huang J, He X (2021a) KEML: A knowledge-enriched meta-learning framework for lexical relation classification. In: Thirty-Fifth AAAI conference on artificial intelligence, AAAI 2021, Thirty-Third conference on innovative applications of artificial intelligence, IAAI 2021, The eleventh symposium on educational advances in artificial intelligence, EAAI 2021, Virtual Event, February 2-9, 2021, AAAI Press, pp 13924–13932, https://ojs.aaai.org/index.php/AAAI/article/view/17640