IET Software

Review Article

Literature survey of deep learning-based vulnerability analysis on source code

ISSN 1751-8806
Received on 11th March 2020
Revised 28th July 2020
Accepted on 29th September 2020
doi: 10.1049/iet-sen.2020.0084
www.ietdl.org

Abubakar Omari Abdallah Semasaba[1] ✉, *Wei Zheng*[1], *Xiaoxue Wu*[2], *Samuel Akwasi Agyemang*[1]
[1]*School of Software, Northwestern Polytechnical University, Xi'an, Shaanxi, People's Republic of China*
[2]*School of Cyberspace Security, Northwestern Polytechnical University, Xi'an, Shaanxi, People's Republic of China*
✉ *E-mail: abu_sem@mail.nwpu.edu.cn*

Abstract: Vulnerabilities in software source code are one of the critical issues in the realm of software code auditing. Due to their high impact, several approaches have been studied in the past few years to mitigate the damages from such vulnerabilities. Among the approaches, deep learning has gained popularity throughout the years to address such issues. In this literature survey, the authors provide an extensive review of the many works in the field software vulnerability analysis that utilise deep learning-based techniques. The reviewed works are systemised according to their objectives (i.e. the type of vulnerability analysis aspect), the area of focus (i.e. the focus area of the analysis), what information about source code is used (i.e. the features), and what deep learning techniques they employ (i.e. what algorithm is used to process the input and produce the output). They also study the limitations of the papers and topical trends concerning vulnerability analysis.

1 Introduction

Vulnerabilities in software significantly undermine the security of computer systems and IT infrastructure of organisations. For instance, vulnerabilities exploited by the WannaCry ransomware have affected a wide range of systems and users worldwide in 150 countries. It is estimated that this ransomware led to over $4 billion worth of losses worldwide [1]. The analysis of vulnerabilities is an expensive and critical process. According to Chowdhury and Zulkernine [2], one of the major causes of breaches in security is vulnerabilities that can be exploited in software. The amount of time that a vulnerability remains undetected makes a system more likely to be exploited by attackers. This can result in organisations or companies suffering from financial losses and irreparable damage to their reputation [3].

Techniques for identifying vulnerabilities in code execution can be categorised into static, dynamic, and hybrid approaches. Static techniques that rely on source code analysis include code similarity detection [4–6], rule-based analysis [7], and symbolic execution [8]. However, this method often struggles to reveal vulnerabilities that might occur during runtime. Dynamic techniques that focus on detecting vulnerabilities that manifest during program execution include taint analysis [9, 10] and fuzzing [11, 12], but in general, they have low-code coverage. In order to overcome the weaknesses mentioned above, hybrid techniques combine both static and dynamic analysis techniques. However, all these approaches rely on a limited set of vulnerability related patterns which raises the challenge of detecting vulnerabilities. Although numerous approaches have been proposed for vulnerability detection, the number of vulnerabilities reported in common vulnerabilities and exposures (CVE) [13] has been increasing each year.

Throughout the years, different methods that pertain to the detection of vulnerabilities have been explored significantly. With the growth of machine learning (ML), opportunities have emerged for intelligent and efficient vulnerability detection. The primary area of operation for these ML-based approaches is on the software source code. Researchers have applied source code-based features, i.e. methods, comments, identifiers, abstract syntax trees (ASTs) [14–19], program dependency graphs (PDGs) and control flow graphs (CFGs) [20], as indicators for identifying potentially vulnerable files or code fragments. Each representation provides a level of abstraction, which creates relationships among elements represented from the source code. These relationships created are explicit and implicit. The importance of each code representation depends on which aspect of focus its software engineering (SE) tasks requirement relates to. An example of this would be the use of comments and identifiers for locating features [21] and the (re) modularisation of software.

ML in conjunction with manually defined or handcrafted features, has been used by many SE researchers for solving important SE tasks [22]. Despite feature selection and the uses of ML-based approaches for these SE tasks, their performance is influenced by the underlying data sets [15]. With the improvement of computational power and an increase in memory brought about by modern computer architectures, new approaches for canonical ML-based tasks have been developed [23]. The rise of deep learning (DL) has paved the way for advancements in several fields. These new approaches employ representation learning to extract useful information from unlabelled software data automatically. The amount of value brought about by DL approaches supports a form of feature engineering [24].

The interpretation of transformations within the data drastically reduces the cost of modelling the code. This is because the code representations store data to aid in improving the learning process's effectiveness. Then, the performance increases since the discovery of data correlations in high-dimensional spaces are interpreted better by learning algorithms at a higher efficiency rate than humans.

It is essential to systematically summarise the empirical evidence obtained on these techniques from the existing literature and studies to facilitate the use of DL for vulnerability detection. This paper reviews all the studies between the period of 2014 and 2020 extensively. To perform this review, we have identified a taxonomy on DL for source code vulnerability detection.

This literature survey aims to summarise, analyse and assess the empirical evidence regarding: (i) DL models for vulnerability detection, (ii) main objectives of the vulnerability detection, (iii) comparisons of the features used for the neural network models, and (iv) summarise the limitations of vulnerability detection. We provide future guidelines and topical trends for software practitioners and researchers regarding source code vulnerability detection. To achieve this aim, we extensively searched through 6 digital libraries and identified 28 studies to answer the research questions on DL for source code vulnerability detection. The primary studies were selected according to the quality assessment of the studies and relevance.

Contributions of this work are as follows:

- A detailed comparative analysis of existing literature on the topic, structured according to the proposed taxonomy, which highlights possible new research directions.
- An analysis of empirical evaluations reported in the papers.
- A summary of different focus areas and how each of the surveyed papers handled and compared aspects of vulnerability analysis given the limitations they had.

Table 1 Research questions for the literature survey

RQ#	Research questions	Motivation
RQ1	which DL algorithm have been used for vulnerability detection on source code?	identify the ML techniques commonly being used in the literature survey
RQ2	which data sets are the most used for vulnerability detection on source code?	identify data sets reported to be used for vulnerability detection and their availability
RQ3	what are the limitations for DL for source code vulnerability detection?	identify limitations faced by the surveyed works
RQ4	what are the features used for DL vulnerability detection on source code?	identify the features used in the reviewed works
RQ5	what is the reporting quality of the existing papers?	analyse the quality of published papers on the basis of our five question criteria

Table 2 Inclusion and exclusion criteria

Include	Exclude
peer-reviewed papers published from 2014 to 2020	papers not published in computer science journals
papers written in English	papers that do not use DL
focus on DL for vulnerability detection	conference papers that present work-in-progress or incomplete research are also excluded
retrieved from digital sources	journal papers that do not present a complete or significant portion of their methodology
if a study was published more than once, we include the more detailed version.	—
conference proceeding papers that present complete research during the same period.	—
papers available in full text	—

Table 3 Digital libraries

Sources	
1.	Google Scholar [25]
2.	IEEE Xplore [26]
3.	ACM Digital Library [27]
4.	Springer Link [28]
5.	Wiley Online Library [29]
6.	Science Direct [30]

Table 4 Keywords used for searching on the digital libraries

Keywords	
1.	software vulnerability detection
2.	source code security bug detection
3.	source code vulnerability detection
4.	source code bug detection
5.	vulnerability detection using DL
6.	vulnerability detection on source code using DL

- A detailed comparative analysis of existing literature on the topic, structured according to the proposed taxonomy, which highlights possible new research directions.
- Quality assessment of the included studies.

The rest of the paper is structured as follows. First, in Section 2, we present the methods for data collection. We then discuss the quality assurance process of the selected papers and the research questions addressed by the review. We present related researches in different areas of ML vulnerability detection in source code (Section 3). We then discuss the background of software vulnerability analysis (Section 4). Then we discuss the results of the research questions and the quality assurance scores (Section 5). We then present the taxonomy we propose to organise reviewed vulnerability analysis techniques for DL (Section 6). We then categorise the papers according to the presented taxonomy (Section 7). We then highlight the issues and challenges (Section 8). Then we discuss the topical trends and how to advance them (Section 9). Finally, we discuss the conclusion of the review (Section 10).

2 Methods

This section details how the papers for the literature survey are collected. The collection aimed to identify suitable papers related to DL on source code from known journals.

2.1 Research questions

This literature survey aims to assess the empirical evidence from the studies using DL techniques in the literature. Table 1 presents five research questions addressed in this literature survey and the motivations behind them.

2.2 Eligibility criteria

A systematic literature search was undertaken to identify suitable papers related to DL on source code from known journals. The papers' search strategy included the identification of search terms and the inclusion and exclusion criteria. Having a defined inclusion and exclusion criteria helps in reducing bias in the research. Mendeley version 1.19.4 was used to manage the bibliographic information of the papers generated through digital sources. Using Mendeley, we screened the papers by reading their titles and abstracts. We removed papers that do not meet the inclusion criteria. The starting date for the peer-reviewed papers was selected as 2014 since we were only able to find related papers starting from that year. The eligibility criteria are listed in Table 2.

2.3 Digital libraries and search strategy

2.3.1 Digital libraries: We conducted an automatic search on six widely used digital libraries in order to find the relevant papers. The digital libraries used for the search are listed in Table 3.

2.3.2 Search keywords: To find the relevant papers for the review, we used a set of search terms in the aforementioned digital libraries. The search keywords used for this study are listed in Table 4. The search was completed on 15 January 2020, papers published after this date are not considered part of this paper. The search resulted in 85 papers which were used for further analysis.

2.4 Study selection

As a first step, duplicate studies (i.e. papers presented in multiple databases) were removed from the initial pool of 85 collected papers, which resulted in the removal of 20 duplicate studies. In the next phase, irrelevant literature was removed by reading the title and abstract of the studies, which resulted in the final set of 24 studies. To reduce the risk of missing work that might be relevant. Through following the guidelines by Wohlin [31], the last author of this paper performed snowballing. Snowballing involves identifying relevant papers from the references (backward snowballing) and citations (forward snowballing) of the selected papers.

The newly identified studies are then filtered based on the inclusion and exclusion criteria. We used Google Scholar to analyse the papers and search for relevant papers present in the selected pool of papers. The process resulted in four more papers being selected, resulting in a total of 28 studies for further analysis. To reduce the bias in selecting the studies, initially, the first author and last author of the paper independently performed the selection. Then the results of both the authors were matched. All inconsistencies in the selection of studies were discussed and resolved in follow up meetings with the other authors.

2.5 Quality assessment

Quality assessments are well-established activities in systematic reviews to help the researchers gain confidence in the results and the conclusions drawn from the papers [32, 33]. The typical process is performed for selecting the relevant papers, the difference in the quality and weighting of each paper, interpretation of the results, or the future recommendations [34]. For this literature survey, we conducted a quality assessment for weighting each of the papers and their future research recommendations. To assess the quality of the papers, we adopted a set of five common quality assessment questions from [34].

Table 5 Quality assessment criteria

Quality assessment number	Quality assessment question
QA1	are the research aims well-articulated?
QA2	is the proposed technique described clearly?
QA3	is the experimental design appropriate?
QA4	is there a clear statement of finding and relate to the aim of the research?
QA5	does the research add value to academia or industry?

Table 5 lists five questions assigned with the possible answers 'Yes,' Partially' or 'No.' The three questions are scored as 1, 0.5, and 0, respectively. Each paper's quality score is computed as the sum of all the scores of the quality assessment questions. For each paper, a quality score is identified. Each paper is evaluated by at least two authors of this study. When there is a conflict in the assignment of quality scores, a discussion would be held among all the authors to reach consensus in review meetings. Table 6. highlights the quality assessment score given for each of the studies.

3 Related work

In SE, various techniques have been investigated for vulnerability detection. Other academic works have addressed other aspects of surveying contributions for ML and vulnerability detection approaches from various perspectives. For this section, different works related to vulnerability detection are studied.

Liu *et al*. [35] studied different code analysis techniques such as static analysis, fuzzing, and penetration testing. Besides that, the authors take different sets of vulnerability analysis methods that align with vulnerability discovery techniques. In this survey, the authors highlight the advantages and disadvantages of the techniques and define future directions for vulnerability discovery. A study by Shahriar and Zulkernine [36] also reviewed vulnerability detection studies based on different code analysis techniques. Their work extensively compares and contrasts different techniques such as static, dynamic, and hybrid analysis. Moreover, their study also discusses three approaches for mitigating program security vulnerabilities: secure programming, program transformation, and patching.

In [37], the authors provide a perspective on vulnerability prediction through feature-based ML. In their work, the authors also sort out related types and define the basis for vulnerability feature definitions. Besides, the authors also compare the different methods for feature generation and analyse the difficulties and challenges in the field. Malhotra [38] presented an extensive

Table 6 Quality assessment scores

Reviewed paper	QA1	QA2	QA3	QA4	QA5	Sum
2	0.5	0.5	1	0.5	0.5	3
4	0.5	1	1	0.5	1	4
5	0.5	0	1	1	1	3.5
6	0.5	0	1	1	1	3.5
7	1	0.5	0.5	0	1	3
8	1	1	0.5	1	0.5	4
9	1	0.5	1	0.5	1	4
12	1	1	0.5	0	1	3.5
14	1	1	1	1	1	5
15	1	1	1	1	1	5
16	1	1	0.5	1	1	4.5
17	1	1	1	1	1	5
18	0	1	1	0.5	0.5	3
19	0.5	1	1	0.5	0.5	3.5
21	0.5	1	0.5	0	1	3
22	0.5	0.5	0.5	1	0.5	3
23	1	0.5	0.5	0.5	0.5	3
26	0	0.5	1	0.5	1	3
27	1	0.5	0.5	0.5	0.5	3
28	1	1	0.5	0.5	0.5	3.5
29	0.5	1	0.5	1	1	4
31	0	0.5	1	0.5	1	3
33	1	1	0.5	0.5	0.5	3.5
34	1	1	1	1	0.5	4.5
80	1	1	1	1	1	5
88	1	1	0.5	1	1	4.5
89	0.5	1	0.5	0.5	1	3.5
90	1	1	0.5	0	1	3.5

review of software defect prediction studies that focused on ML techniques. In their study, ML techniques and their performances were studied. Liu *et al*. [35] generally reviewed software vulnerability detection studies that use code analysis and ML techniques, including static analysis, fuzzing, and penetration testing.

Ghaffarian and Shahriari [39] provided a review of the studies which focused on ML for vulnerability detection. the reviewed studies focused on those with their features extracted from software metrics, the standard definition of vulnerability patterns, and patterns of anomalies. Jie *et al*. [40] wrote a similar survey that builds on the summary of these studies; however, they proposed a vulnerability analysis framework. Allamanis *et al*. [41] provided a review from the perspective of comparing natural languages with programming languages and discussing the motives of the model design. These motives were based on the similarities and differences of the languages and code. Lin *et al*. [42] provided a survey of the literature on neural networks and DL-based approaches for vulnerability detection. Their survey focuses on code semantic understanding for vulnerability discovery.

This paper focuses on DL for vulnerability detection on source code and highlights the DL techniques employed for each of the selected studies. Moreover, we provide a taxonomy that highlights factors such as objectives, features, and algorithms used in vulnerability analysis for source code.

4 Background of software vulnerability analysis

4.1 Definition

Software vulnerabilities (vulnerabilities in short) are briefly defined as ‘software bugs that have implications on security’ [43]. Vulnerabilities can be viewed as a subset of software defects [44]. When vulnerabilities are exposed and exploited by attackers, it can lead to security failures due to policy violations. In [39], vulnerabilities are defined as: a software vulnerability is an instance of a flaw caused by mistake in the design, development, or configuration of software such that it can be exploited to violate some explicit or implicit security policy.

4.2 Conventional approaches

The problem of software vulnerability detection has led to several approaches studied and investigated by the academic community. All research efforts try to propose improved approaches in comparison to previous works. These efforts propose improvements in vulnerability coverage, the precision of discovery and so on. Regarding vulnerability detection in source code, among the many different approaches, some are more established in the software industry, namely.

4.2.1 Static code analysis: In static code analysis, software or program is analysed without the need to execute it. Static approaches utilise an abstraction that is generalised to analyse the properties of a program [39]. Static analysis is sound and greatly depends on the accuracy of the generalisation. The more accurate it is, the fewer the reported false vulnerabilities. Based on this, a trade-off between precision and computational efficiency. Examples of such methods include data flow analysis [45], symbol execution [46], rule/template-based analysis [47], and theorem proving [48].

4.2.2 Dynamic code analysis: In dynamic code analysis, programs are analysed by executing a program using specific inputs and monitoring its behaviour at runtime [39]. In this approach, test cases are used to analyse program properties. Since there is a large variety of inputs and runtime states, the dynamic analysis does not thoroughly analyse the entire program's behaviour. For this reason, dynamic analysis can be complete but cannot be sound, since some vulnerabilities can be found from unseen program states. Dynamic analysis includes fuzz testing [47] and taint analysis [10, 49] and often have issues with low code coverage.

4.2.3 Hybrid analysis: In a hybrid analysis, programs are analysed by a mixture of static and dynamic analysis techniques. Despite hybrid analysis having the benefits of both static and dynamic analysis, these analysis approaches are still not sound and complete. This is due to hybrid analysis approaches suffering from the limitations of both approaches [39].

4.3 ML-based vulnerability detection

Several ML-based approaches have been introduced to help propose various features for detection systems. In [39], several different approaches that apply ML were categorised. The authors categorised them into three main categories.

4.3.1 Vulnerable code pattern based: This category of ML-based vulnerability detection involves the use of ML-based approaches (mostly supervised) to extract vulnerable code patterns from many vulnerable code samples. Then pattern matching techniques are used to detect and locate vulnerabilities found in the source code. Some studies that used text mining-based techniques for extracting code patterns from code. Textual tokens from the source code show these patterns (i.e. the bag of words technique [50], which is used to convert tokens of source code into vectors). Other techniques, such as *N*-gram, were used for mining the program source code [51]. However, some researchers have argued that text mining methods did not capture the semantic meanings of the source code because counts were based on term co-occurrences, and this did not preserve contextual dependencies of the code [52].

Several studies used static code analysis tools to extract structured feature sets as code patterns for ML algorithms to learn from [42]. The feature sets were extracted from different program representations, which are generated through static analysis. Examples of such features include ASTs, CFGs, PDGs, data flow graphs (DFGs), and so on. Compared to software metrics and frequency-based code features, these features generated from code analysis tools reveal more information based on its program representation, which provides a view of source code from different focus perspectives. For example, ASTs represent source code in a tree view structure. On the other hand, PDGs, DFGs, and CFGs analyse variable flows from source to the sink to understand program structures and building dependencies. Yamaguchi *et al*. [20] combined ASTs, PDGs, and CFGs into a joint representation known as code property graphs (CPGs) to provide a detailed graph structure for code properties. The extraction of feature sets based on source code structures on a high level can reflect how vulnerabilities can be found.

4.3.2 Software metrics-based: Several studies utilise software metrics as the feature set to build prediction models. These studies use the model to assess the vulnerability status of software artefacts based on software metrics. Such metrics include McCabe [53], and Code Churn [54], which are numerical indicators for the quality of software products from different perspectives. McCabe metrics are software complexity metrics, whereas code churn is used as an indicator of how frequently modified code has a tendency to be erroneous. Despite their long history of use in vulnerability prediction, they are not direct indicators of vulnerabilities. This led to many studies that relied on this method as features and reported poor vulnerability detection results [39].

4.3.3 Anomaly based: Early studies such as [55, 56] applied rules from individuals as a knowledge template for detecting vulnerable code that did not conform to programming guidelines. Later studies applied features extracted from function calls [23], imports, and API usage patterns [57] for vulnerability detection. These feature sets are usually related to small sets of vulnerabilities, thus limiting their capabilities. For example, features from imports or function calls can train classifiers that detect vulnerabilities by header files, and API usage symbols are related to missing checks to check for vulnerabilities caused lack of validation or boundary checks. Hence, solutions proposed by these studies may be suitable for task-specific applications [42].

4.4 Deep learning techniques

ML models' application for vulnerability detection tasks relies on the extraction of handcrafted features from static or dynamic code analysis to help the model learn the vulnerable code patterns. Deep neural networks (DNNs) have been used for several purposes, such as image recognition [58] and machine translation [59]. Among the reviewed studies, different network structures are used.

4.4.1 Unsupervised pre-trained networks (UPNs): UPNs are used as a means for training deep feed-forward neural networks through the use of a discriminative neural net, which initialises from one which is trained from a supervised criterion. The main reason for using such networks was the difficulty of local optima in the objective functions and the proneness of models to overfitting [60]. An example of a UPN that is commonly used is variational autoencoders (VAEs) [61]. Hence, UPNs have a low proneness to overfit, which motivates researchers to apply UPNs to learn vulnerable code semantics.

4.4.2 Convolutional neural networks (CNNs): CNNs: convolutional neural networks are a type of neural network which learns higher-order features in the data through convolutions. They are well suited to solving problems that pertain to object recognition and image classifications. CNNs are beneficial in the context where a problem that needs to be solved can be previewed like an image. Recently, CNNs have been applied in natural language processing (NLP) and have fostered exciting results [62]. CNN's excel when it comes to learning the spatial structure of input data. In the CNN structures, there are several hyperparameters, such as the numbers and sizes of convolution filters, pooling strategies, functions of activation, and so on. CNN's are designed to learn structured data with a spatial nature [63]. The first layer of the CNN learns features from semantically similar pixels that are nearby. The succeeding layers learn features at a higher level, used by the subsequent layers (i.e. dense layers) for classification. CNN's have been widely adopted to facilitate NLP tasks due to their capability to learn features from nearby pixels. In tasks such as text classification, CNN filters are applied using context filters (i.e. word embeddings), which project words within the context window to vectors of semantically similar words within proximity [64]. This allows CNNs to capture the contextual meanings of words, which motivates researchers to apply it for learning the semantics of vulnerable code.

4.4.3 Recurrent neural networks (RNNs): RNNs belong to a family of feed-forward neural networks that are different from other networks as they send information over time-steps. They have an advantage over other networks as they allow for parallel and sequential computation; they can compute anything like a traditional computer. RNNs are known to have similarities with the human brain, which is like an extensive feedback network of connected neurons that somehow can translate lifelong sensory input streams like a sequence of motor inputs. Compared to other feed-forward networks RNNs are designed for the processing of sequential data. Therefore, variants of RNNs are applied in several studies for capturing long-term dependencies of a sequence. For this reason, bidirectional long short-term memory (BLSTM) [65] networks and gated recurrent units (GRUs) [66] are often used to learn the contextual dependencies which are crucial for understanding the semantics of many types of vulnerabilities [39].

5 Results and discussion

This section presents the results obtained from the primary studies selected for the literature survey. First, we present an overview of each paper along with its corresponding ranking based on its quality assessment score. Secondly, the answers to the research questions are provided in each sub section.

5.1 Quality assessment scores

Table 6 provides a detailed listing of each quality criterion and the corresponding score for each included study.

5.2 Research questions

5.2.1 RQ1: which DL algorithms have been used for vulnerability detection on source code?: Based on the reviewed studies, DL algorithms are characterised as follows:

- Convolutional neural network (CNN).
- Long short-term memory (LSTM).
- Bidirectional long short-term memory (BLSTM).
- Multilayer perceptron (MLP).
- Gated recurrent unit (GRU).
- Bidirectional gated recurrent unit (BGRU).
- Graph neural network (GNN).
- Deep neural network (DNN).
- Bidirectional recurrent neural network (BRNN).
- Contextual long short-term memory (CLSTM).
- Artificial neural network (ANN).

5.2.2 RQ2: which data sets are the most used for vulnerability detection on source code?: A variety of data sets have been used in the literature survey. The data sets used by the studies can be publicly available or not shared. For the data that is not shared, it is challenging to verify the results on such data sets, and such studies are not repeatable. The significant data sets used are characterised as follows in the order of the most used:

- National Vulnerability Database (NVD) and Software Assurance Reference Data set (SARD).
- LibTiFF, LibPNG, FFmpeg, Pidgin, VLC Media Player, Asterisk.
- NIST Juliet Data set.

0.2.3 RQ3: what are the limitations for DL for source code vulnerability detection?: Based on the reviewed studies, the limitations of the studies are characterised as follows:

- focus single library/API function calls;
- single programming language focus;
- use of function names to represent functionalities;
- labelling of the functions;
- use of other raw data from online;
- use of single metrics for result analysis;
- training time;
- data imbalance;
- vulnerability detection granularity;
- data set size.

5.2.4 RQ4: what are the features used for DL vulnerability detection on source code?: Upon analysing the reviewed studies, the features used are as follows:

- Abstract syntax trees (ASTs).
- Code gadgets.
- Control flow graphs (CFGs).
- Code property graphs (CPGs).
- Code tokens.
- Lexed representation of source code.
- Code slices.
- Function calls.
- *N*-gram analysis.

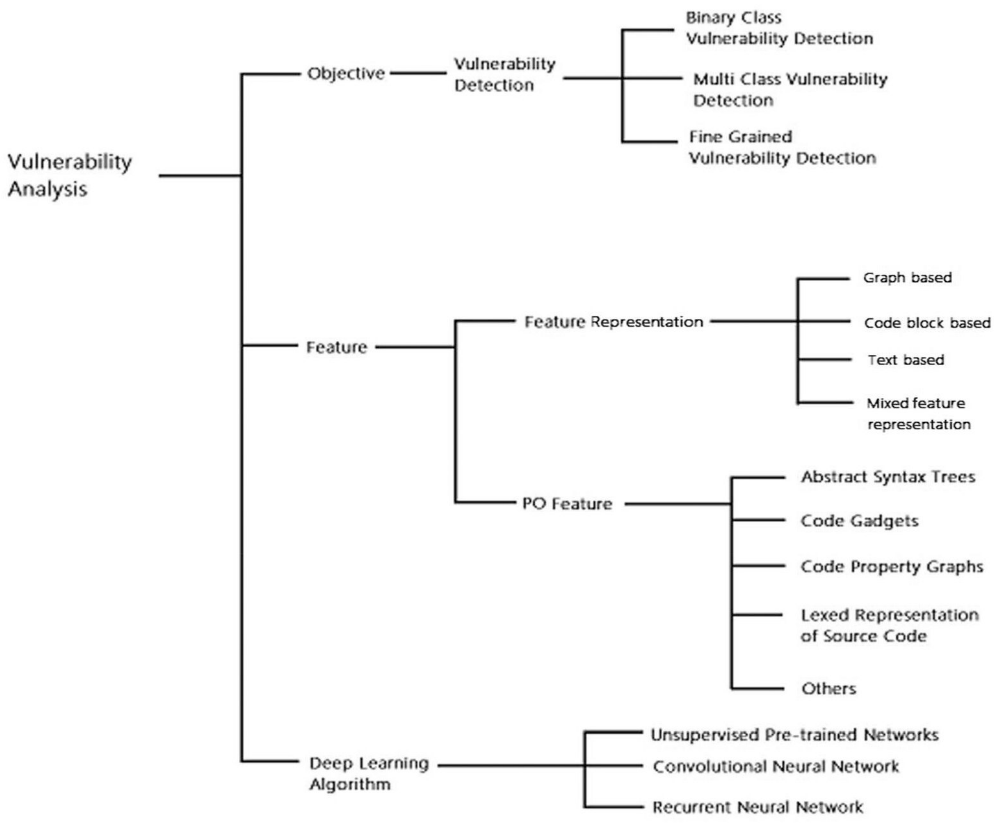

Fig. 1 *Taxonomy of DL techniques for vulnerability analysis on source code*

6 Taxonomy of deep learning techniques for source code vulnerability analysis

This section introduces a detailed taxonomy on how DL is used for vulnerability analysis in the reviewed papers. Upon analysing the papers, we identify four significant dimensions by which the surveyed works can be organised descriptively. The first dimension characterises the final objective of the analysis concerning vulnerability detection, e.g. binary class vulnerability detection. The second dimension describes the features of the analysis and how the features are represented, e.g. through graph-based representation, and what features are considered, e.g. ASTs. The final dimension is how DL algorithms are implemented for the analysis, e.g. ANNs.

Fig. 1 shows a graphical representation of the taxonomy. The rest of this section is structured according to the taxonomy. Section 6.1 describes the objective dimension; features are described in Section 6.2, and DL algorithms are described in Section 6.3.

6.1 Vulnerability analysis objectives

Vulnerability analysis generally demands the ability to detect (or predict) vulnerabilities, which may result in security risks. However, as the focus centres on source code, the primary objective of vulnerability analysis is to detect or predict patterns of vulnerabilities. This prediction is conducted through the gathering of knowledge from past circumstances and learning from them. For source code, vulnerabilities are predicted by identifying the potential that a source code file may have in terms of being vulnerable. Indeed, all the reviewed works covered vulnerability detection in some form, as the primary objective. Depending on the DL technique used to achieve this, the overall output can help analysts understand whether a sample needs further inspection. We find three different vulnerability detection variations as an objective: Binary class vulnerability detection, multiclass vulnerability detection, and fine-grained vulnerability detection.

6.1.1 Binary class vulnerability detection: This type of vulnerability detection is the most common vulnerability detection strategies studied amongst several of the papers used for the survey. This form of vulnerability detection involves identifying whether a set of source code files used for a test is vulnerable or not. Recognising whether a source code file is vulnerable or not is highly significant as it helps testers focus their tests on catering to these files, which have a high possibility of being vulnerable. Also, this objective has been studied in the literature, and several reviewed papers target identifying whether a set of source code files are vulnerable or not [14, 16, 18, 19, 60, 67–76]. Considering a large number of vulnerable source code that is available, the significance of recognising which files are vulnerable can help reduce the workload on testers.

6.1.2 Multiclass vulnerability detection: Analysts might be interested in identifying much more than just whether a set of source code files are vulnerable but also multiple classes of vulnerabilities. The classes of vulnerabilities determine that type of vulnerability is essential since this information gives analysts an idea which principles can be used to pin down the precise location of vulnerability quickly and reduce the workload required in terms of fixing it. Multiclass vulnerability deals with recognising the different types of vulnerabilities that might be present within the source code files. Among the surveyed papers, only one paper deals with multiclass vulnerability as its main objective [77].

6.1.3 Fine-grained vulnerability detection: Fine-grained vulnerability detection involves the localisation of vulnerabilities within the source code files. This is used to pinpoint the lines of codes within a function which have the highest probability of being vulnerable. Fine-grained vulnerability detection operates on a courser granularity with a focus on the function level typically. The discovery of these lines of code with a higher accuracy allows for the focus on those specific code lines since the probability of its vulnerability is higher. This vulnerability aspect is focused on by several of the surveyed papers [15, 17, 78–87] Table 7.

6.2 Vulnerability analysis features

This section covers the features of samples that are considered for the analysis. Section 6.2.1 reports on how the features are represented from the source code input, and Section 6.2.2 reports on the which features are taken into account for consideration.

6.2.1 Feature representation: Upon reviewing the studied works, we decided that four types can best represent the features. The rationale behind the proposed categorisation is based on is how the source code has been processed to generate feature representations that facilitate the DNNs understanding of code semantics and patterns from vulnerable code snippets.

(a) *Graph-based feature representation:* The graph-based representation of source code is done through the use of parsers, which can extract the information from the source codes. A large body of studies applies feature representations through the use of graph-based program representations. Examples include ASTs, CFGs, PDGs, or a combined representation of graph-like structures. Some examples of tools used to extract graph-based representations from the source code files include Joern [71, 80, 87] and CodeSensor [14, 15, 19]. Graph-based representations of source code use some form of abstract representation that preserves the semantic information along with the syntax. Several of the reviewed papers use this form of feature extraction due to the semantic benefits. Some of the reviewed works use a single form of graphical representation for their feature extraction [14–19, 71, 73, 75, 80, 87]
(b) *Code block-based feature representation:* For code block-based feature representation, studies under this category utilise DNNs for extracting feature representations from sequential code entities such as function calls, code snippets, code gadgets, and so on. Some of the reviewed papers rely on the use of code block-based representations of source code [67, 70, 72, 74, 76, 77, 82, 83, 85]
(c) *Text-based feature representation:* For this category of feature, representations are learned directly from the source code text surface. Examples include Lexed code representation and using the source code files directly. Among the reviewed papers, several rely on the use of text-based representations of source code [60, 67, 68, 78, 79, 86]
(d) *Mixed features-based representation:* This category includes recent studies that combined the abovementioned three types of feature representations. Among the reviewed papers, several rely on the use of mixed feature-based representations of source code [19, 69, 81, 84].

6.2.2 Portable output features: This section discusses the features which are used by the reviewed papers in order to achieve the objectives which are outlined in Section 6.1.

(a) *Abstract syntax trees:* An AST represents the functional structure of code, such as expression, scopes, and declaration. Simultaneously, redundancies are avoided by omitting unnecessary syntactic details [88, 89], which do not affect the original function of the code even after the parsing process. These include whitespaces, punctuation marks, or comments. Analysing specific nodes of the AST can provide a detailed picture of a code structure relevant to understanding the semantics. Among the reviewed works, the majority rely on ASTs as the main representation of the source code to extract the information as portable output features [14–19, 73, 75]. For the generation of ASTs tools such as CodeSensor [14, 15], C/C++ source code files, and Esprima [18] for JavaScript files in several of the reviewed works.
(b) *Code gadget:* A code gadget is composed of several program statements such as lines of code, which are semantically related to each other in terms of data or control dependency. The primary use of code gadgets as a portable output feature comes from its ability to represent the best dependencies of source code programs, making it suitable for inputs into neural networks. A few reviewed works rely on code gadgets as the features used for vulnerability detection [19, 77, 83, 85]. However, since [83] is a further extension to [85]. The use of code gadgets is adopted together with the use of code attention, which is inspired by the notion of region attention to classify the type of vulnerabilities.
(c) *Code property graphs:* Vulnerability discovery through the utilisation of CPGs takes into account the code dependencies, structure, and control flow. CPGs merge the concepts of classical program analysis, which consists of ASTs, CFGs, and PDGs into a joint data structure [20]. The main idea behind the use of CPGs as a portable output feature is capturing the characteristics brought about by the representations mentioned above and offering the advantages as a single representation. To enhance the semantic information that can be gathered by code representations as features, a few of the surveyed works utilise CPGs [71, 80]. In order to generate CPGs, tools such a Joern [71, 80, 87] are utilised for C/C++ code.
(d) *Lexed representation of source code:* Lexing is a process which involves the use of a lexer in order to generate useful features from

Table 7 Characterisation of surveyed papers having fine-grained vulnerability detection as the objective

Paper	Algorithms	Features	Data set name	Data set size	Data set shared
[82]	BLSTM	operational semantics features	CVE	8200 for Python, 8325 from C/C + +	Yes
[78]	Memory Network	source Code Files	Juliet test suite	10,000	Yes
[84]	BLSTM	PDG, Code Slices	NVD and SARD	15,592	Yes
[80]	ANN	CPG	CWE-119, CWE-399	28,049	Yes
[15]	BLSTM	AST	LibTiFF, LibPNG, FFmpeg, Pidgin, VLC Media Player, Asterisk	LibTiFF 873 functions, LibPNG 542 functions, FFmpeg 5112 functions, Pidgin 8079 functions, VLC Media Player 3678 functions, Asterisk 14,704 functions	Yes
[83]	MLP, CNN, LSTM, GRU, BLSTM, BGRU	Code Gadgets	NVD and SARD	368 from NVD, 14,000 from SARD	Yes
[85]	BLSTM	Code Gadgets	NVD and SARD	10,691	Yes
[17]	BRNN-vdl	AST	NVD and SARD	382 from NVD, 9864 from SARD	Yes
[79]	BLSTM	Bag of Characters (Centred and Uncentred)	StackOverflow	1792 unique Characters	No
[87]	GNN	Graph Embedding of Code	Linux Kernel, QEMU, Wireshark, FFmpeg	Linux 12811, FFmpeg 13962, Qemu 11910, Wireshark 10004	Yes
[86]	RNN, CNN	lexed representation of code	SATE IV Juliet Test Suite, Debian Linux distribution, GitHub	SATE IV 121353, GitHub 9706269, Debian 3046758	No
[81]	Word2Vec + CNN	CFG, lexed representation of code	GitHub, Debian, Source	Source 981924, GitHub 333024, Debian 569360	No

raw source code. In lexing, critical tokens are captured from source code functions while the representation is kept generic and the total vocabulary size is minimised. There are several advantages to this representation of source code as a portable output feature as it empowers transfer learning across data sets. Some of the surveyed works use this form of representation [81, 86]. However, in [86], the lexed source code representation is used in conjunction with CFGs as a form of build based feature extraction.

(e) *Others:* Among the reviewed works, several other forms of source code representation are utilised as portable output features. Such feature source representations include utilising the source code files themselves [78], representing the source code as code tokens [67, 74, 76], dividing source code files into code slices [72, 84] and using function calls [70]. Besides, other forms of representation include the use of vectors such as binary feature vectors [60] and bags of characters [79]. Furthermore, forms of representation also utilised in the surveyed works include representing source codes in a graph-based structure such as CFGs [81], PDGs [84], and embedded graphs [87]. The extraction of CFGs is done using a tool called Clang and LLVM [81]. Representation of the surveyed works is also done through utilising the program execution paths [69], *N*-gram analysis [68], and operational semantics features [82].

6.3 DL algorithms

This section introduces the DL algorithms used in the surveyed works by organising them based on whether the learning can be categorised as using unsupervised pre-trained networks (Section 6.3.1), convolutional neural networks (Section 6.3.2) or recurrent neural networks (Section 6.3.3).

6.3.1 Unsupervised pre-trained networks: The UPNs used in the reviewed papers are the deep autoencoder in [60]. However, the deep autoencoder used is a variant since it utilises a mechanism of stacking and denoising in order to enhance the vulnerability prediction capacity.

6.3.2 Convolutional neural networks: Among the reviewed papers, several of them utilise CNNs as the DNN model [70, 71, 75, 81, 83, 86].

6.3.3 Recurrent neural networks: Among the reviewed works several of them use some form of RNNs. classical RNNs [86], long short-term memory (LSTM) [16, 67, 69, 70, 73, 76, 83], contextual long short-term memory (CLSTM) [72], bidirectional recurrent neural networks (BRNNs) [17, 74], bidirectional gated recurrent unit (BGRU) [83], and bidirectional long short-term memory (BLSTM) [14, 15, 19, 77, 79, 82–85].

6.3.4 Others: Other DL models used by the surveyed papers include gated recurrent unit (GRU) [83], multilayer perceptron (MLP) [83], deep neural network (DNN) [68], graph neural network (GNN) [87], Doc2Vec [18] and artificial neural network (ANN) [80].

7 Characterisation of surveyed papers

In this section, we characterise each of the reviewed works based on their vulnerability detection objective.

7.1 Multiclass vulnerability detection

The lists of all the reviewed works which have multiclass vulnerability detection as the main objectives are listed in Table 8.

7.2 Binary class vulnerability detection

Table 9 lists all the reviewed works which have binary class vulnerability detection as their objective.

7.3 Fine-grained vulnerability detection

Table 7 lists all the reviewed works which have fine-grained vulnerability detection as their objective.

8 Issues and challenges

Based on the characterisation that has been detailed in Section 7, this section shall explain some of the challenges and issues documented in the surveyed papers. Some of the problems highlighted by a majority of the papers are focus on a single programming language (Section 8.1), focus on libraries/API function calls for vulnerabilities (Section 8.2), and the data set used (Section 8.3).

8.1 Single programming language

The focus of a majority of the surveyed works conducts vulnerability detection on data sets with a single programming language. This issue has been highlighted by several reviewed papers as a limitation that can be explored in the future. On several of the surveyed works the programming language of the data set focus ranges from C/C++ [14, 15, 17, 19, 69–72, 77–87] and JavaScript [18, 60, 79]. Having a data set that consists of multiple programming languages would require further exploration.

8.2 Library/API function calls for vulnerabilities

The focus on libraries/API function calls have been documented as a limitation for several reviewed papers [16, 77, 83, 85]. Through the use of libraries/APIs for vulnerability detection, the vulnerability detection granularity is less accurate. Thus, making it more difficult to pinpoint the exact location of vulnerabilities within the functions.

8.3 Data sets

Several of the reviewed works use a variety of data sets that represent both vulnerable and non-vulnerable functions. These data sets which are collected each have a programming language of focus. The works of [15, 17, 19, 73, 75–78, 80, 82–85, 87] made their data sets publicly available for further research. For a large majority of reviewed the papers, the data set's size did not present an issue. This was because many authors were able to meet their objectives with the data set used. However, the work of Wu *et al.* [70] does mention the data set size as one of the limitations in their work.

A common issue found with most of the reviewed works is data set imbalances [15]. This attributed to some biases in the results. However, attempts at handling this imbalance, the use of imbalance strategies would be more prevalent in further improving the prediction results. Furthermore, the proper labelling of functions and data sets also prove a limitation for [81, 86] because of the manual labelling of the functions in both works, which deserves more attention in future works.

In terms of the data set used, the names and sizes are referenced in Tables 7–9. Since the data sets used for the evaluation are rarely shared or made publicly available, it is nearly impossible to compare the works accurately in terms of relevant metrics. Given this lack of reference data sets, we propose three desiderata for vulnerability analysis:

- The data set should be well balanced to prevent biases in the results.

Table 8 Characterisation of surveyed papers having multiclass vulnerability detection as the objective

Paper	Algorithms	Features	Data set name	Data set size	Data set shared
[77]	BLSTM	code gadgets, code attention	multiclass vulnerability data set	33,409 programs	Yes

- The labels for each of the functions need to be improved, such as those from dynamic analysis tools to make the scores from the DL model more complementary with static analysis tools [86].
- Samples should be regularly updated and maintained over the years through publicly available repositories. For example, GitHub (if the data set is made publicly available). Since these tracked changes or modifications can be relevant to the works of future researchers.

9 Topical trends

This section outlines different potential future research directions in vulnerability analysis in source code, i.e. topics that are currently being investigated but have not reached a maturity level.

9.1 Finer grained vulnerability detection

This research area involves identifying the precise location of the vulnerabilities in the source code files or functions. Finer-grained vulnerability detection includes identifying attributes of vulnerabilities and analysing them on a higher granularity. This broadens the scale of vulnerability by identifying specific areas of the source code that might be vulnerable [77, 84]. Another proposed research area among several papers is the ability to pinpoint the location of the vulnerability [16, 77, 83, 85]. This is conducted by detecting potentiality vulnerable functions or a set of lines of code that have a higher probability of being vulnerable. Knowing and pinpointing the exact location of vulnerabilities can be very helpful for finer-grained vulnerability detection.

9.2 Enhancement of rich feature extraction

Another area of focus would be the enhancement of rich feature extraction and feature description. This is related to achieving higher precision and recall in vulnerability analysis [82, 87]. This area is more focused on feature engineering, which is domain specific. This direction shall deal with specific features that are relevant within its application domain. It can help preserve semantic correctness by selecting the best features from a feature set. Other areas relevant for future research include studying how data set labelling, imbalance handling, and rich feature extraction can be used to improve vulnerability detection.

9.3 Cross-project learning

The investigation into cross-domain algorithms for vulnerability analysis in scenarios that address cross-project learning [15]. This area involves the use of cross project-based learning approaches such as transfer learning in order to help enhance the learning time for the DL models. This area involves the use of pre-trained models to train a classifier or fine-tuning the pre-trained model keeping learned weights as initial parameters.

9.4 Enhanced DL models

The use of enhanced DL models for the detection of vulnerabilities is another area of focus for future research. This involves using enhanced or combined DL models for training in testing. The use of enhanced models can help identify possible attributes between models that can help enhance the accuracy of the analysis results. This is especially achieved in the training and testing phases. An example of enhanced DL is BRNN [74].

Table 9 Characterisation of surveyed papers having binary class vulnerability detection as the objective

Paper	Algorithms	Features	Data set name	Data set size	Data set shared
[14]	BLSTM	AST	LibTIFF, LibPNG; FFmpeg	6486 functions	no
[16]	LSTM	AST	20 popular Android applications from 2011, Firefox	26,352 source files	no
[18]	Doc2Vec	AST-JS	JS codes	5024 JS codes	no
[19]	BLSTM	AST, code gadgets	Asterisk, FFmpeg, LibPNG, CWE119, CWE399	81,996	yes
[70]	CNN, LSTM and CNN-LSTM	function calls	N/A	9872 binary programs	no
[72]	CLSTM	code slices	Software assurance reference data set	23,185 programs	no
[68]	DNN	*N*-gram analysis	BoardGameGeek, Connectbot,CoolReader, AnkiDroid	BoardGameGeek 70 classes, Connectbot 51 classes, CoolReader 51 classes, AnkiDroid 28 classes	no
[69]	LSTM	program execution path	NIST SARD, GitHub, exploit-DB	– NIST SARD 26,080 programs – GitHub 560 programs – Exploit-DB 1039 programs	no
[60]	stacked denoising auto-encoder	binary feature vectors	VX Heaven, malicious website labs	27,103	no
[67]	LSTM	code tokens	crosswords, contacts, browser, desk clock, calendar, AnkiAndroid, mms, Boardgamegeek, Gallery, Connectbot, Quicksearchbox, Coolreader, Mustard, K9, Camera, Fbreader, Email, Keepassdroid	15,301	no
[71]	CNN	CPG	SARD	265,190	no
[73]	LSTM	AST	SARD, FFmpeg, LibTIFF, LibPNG, VLC, Pidgin, and Asterisk.	SARD: 136000 files, FFmpeg: 5914 files, LibTIFF: 827 files, LibPNG: 620 files, VLC: 8079 files, Pidgin: 3678 files, Asterisk: 14,704 files	yes
[74]	BRNN	code tokens	FFmpeg, LibTIFF, LibPNG, VLC, Pidgin, and Asterisk.	Ffmpeg: 5614 files, LibTIFF: 776 files, LibPNG: 594 files, VLC: 5573 files Pidgin: 8310 files Asterisk: 8848 files	no
[76]	TCNN	AST	open judge (OJ) system	N/A	yes
[75]	LSTM	code tokens	NIST Juliet data set	44,495	yes

9.5 Impact of multiple programming language data sets

This research direction involves using multiple programming language data sets for a DL model and studying their impact on different metrics for vulnerability detection. This direction of vulnerability detection was studied in the work of [79, 82]. The authors explored the viability of this approach by using operational semantics of a data set of C/C++ and Python files. The experimental results showed that their tool handled both representations of source code files in each respective programming language.

10 Conclusion

This paper presents a systematic literature review of software vulnerability analysis on source code using DL. To sum up, a total of 28 research papers that use DL for vulnerability analysis on source code are reviewed. The reviewed papers are compared and analysed based on various essential factors, including the analysis objective, the DL algorithm, the characteristics of the data set, and the feature representation. There are four main contributions of our work.

First, we provide a detailed description of the reviewed works classifying them based on three dimensions: (i) the objective of the vulnerability analysis; (ii) the DL algorithms used to process these features; and (iii) the vulnerability analysis features. Such a characterisation of DL for vulnerability detection on source code provides an overview of how DL algorithms can be employed in vulnerability analysis, which objectives were achievable, and what features are used.

Secondly, the existing literature is organised according to a taxonomy that detailed the surveyed works, comparing them on different aspects. Each of the papers is grouped by considering the features, DL algorithm, and data sets. The features are classified according to their representation: (i) graph-based feature representation; (ii) code block-based feature representation; (iii) text-based feature representation; and (iv) mixed features-based representation.

Thirdly, we highlighted the main issues with DL in vulnerability analysis: (i) single programming language; (ii) focus on libraries/API function calls for vulnerabilities; and (iii) the data set used.

Fourthly, we identified topical trends that highlighted possible research directions that yield a more significant benefit in exploration and which have already gained traction in the space of vulnerability analysis on source code.

Noteworthy directions regarding research can be investigated further and linked to the contributions mentioned above. The combinations towards the objectives, features, and algorithms can be investigated further to enhance the accuracy and scores on the relevant metrics. The overview of some of the issues highlighted by the surveyed works can provide further ideas that are worth exploring.

11 References

[1] 'Ransomware WannaCry: All you need to know | Kaspersky.' Available at https://www.kaspersky.com/resource- center/threats/ ransomware-wannacry, accessed 03 October 2019

[2] Chowdhury, I., Zulkernine, M.: 'Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities', *J. Syst. Archit.*, 2011, **57**, (3), pp. 294–313

[3] Shin, Y., Meneely, A., Williams, L., *et al.*: 'Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities', *IEEE Trans. Softw. Eng*, 2011, **37**, (6), pp. 772–787

[4] Kim, S., Woo, S., Lee, H., *et al.*: 'VUDDY: A scalable approach for vulnerable code clone discovery'. Proc. 2017 IEEE Symp. Security Privacy, San Jose, CA, USA, May 2017, pp. 595–614

[5] Jang, J., Agrawal, A., Brumley, D.: 'Redebug: finding unpatched code clones in entire OS distributions'. Proc. 2012 IEEE Symp. Security Privacy, San Francisco Bay Area, CA, USA, May 2012, pp. 48–62

[6] Sajnani, H., Saini, V., Svajlenko, J., *et al.*: 'SourcererCC: scaling code clone detection to big-code', *Proc., Int. Conf. Softw. Eng.*, 2016, **1**, pp. 1157–1168

[7] Moohun, L., Sunghoon, C., Changbok, J., *et al.*: 'A rule-based security auditing tool for software vulnerability detection'. Proc. – 2006 Int. Conf. Hybrid Information Technology (ICHIT 2006), Daejeon, Republic of Korea, August 2006, vol. 2, pp. 505–512

[8] Li, H., Kim, T., Bat-Erdene, M., *et al.*: 'Software vulnerability detection using backward trace analysis and symbolic execution'. Proc. – 2013 Int. Conf. Availability, Reliability and Security (ARES 2013), University of Regensburg, Germany, September 2013, pp. 446–454

[9] Cao, K., He, J., Fan, W., *et al.*: 'PHP vulnerability detection based on taint analysis'. 2017 6th Int. Conf. on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), Amity University Uttar Pradesh, Noida, India, September 2017, pp. 436–439

[10] Newsome, J., Song, D.: 'Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software'. Proc. of the 12th Annual Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, February 2005

[11] Dai, H., Murphy, C., Kaiser, G.: 'Configuration fuzzing for software vulnerability detection'. Proc. – 2010 Int. Conf. Availability, Reliability Security (ARES 2013), Krakow, Poland, February 2010, pp. 525–530

[12] Wang, T., Wei, T., Gu, G., *et al.*: 'Taintscope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection'. Proc. of the 31st IEEE Symp. on Security and Privacy (SP '10), Berkeley/Oakland, CA, USA, May 2010, pp. 497–512

[13] 'Common Vulnerabilities Exposures (CVE)', Available at http://cve.mitre.org/, accessed 03 October 2019

[14] Lin, G., Zhang, J., Luo, W., *et al.*: 'Poster: vulnerability discovery with function representation learning from unlabelled projects'. Proc. of the ACM Conf. on Computer and Communications Security, Dallas, Tx, USA, 2017, pp. 2539–2541

[15] Lin, G., Zhang, J., Luo, W., *et al.*: 'Cross-project transfer representation learning for vulnerable function discovery', *IEEE Trans. Ind. Inf.*, 2018, **14**, (7), pp. 3289–3297

[16] Dam, H.K., Tran, T., Pham, T.T.M., *et al.*: 'Automatic feature learning for predicting vulnerable software components', *J. LATEX Cl. FILES*, 2015, **14**, (8), pp. 1–19

[17] Li, Z., Zou, D., Xu, S., *et al.*: 'Vuldeelocator: a deep learning-based fine-grained vulnerability detector', 2020, (2), pp. 1–15

[18] Ndichu, S., Kim, S., Ozawa, S., *et al.*: 'A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors', *Appl. Soft Comput. J.*, 2019, **84**, (105721), pp. 1–11

[19] Ban, X., Liu, S., Chen, C., *et al.*: 'A performance evaluation of deep-learnt features for software vulnerability detection', *Concurr. Comput. Pract. Exp.*, 2018, **31**, (19), pp. 1–10

[20] Yamaguchi, F., Golde, N., Arp, D., *et al.*: 'Modeling and discovering vulnerabilities with code property graphs'. Proc. – IEEE Symp. on Security and Privacy, San Jose, CA, USA, May 2014, pp. 590–604

[21] Dit, B., Revelle, M., Gethers, M., *et al.*: 'Feature location in source code: A taxonomy and survey', *J. Softw. Evol. Process.*, 2013, **25**, (1), pp. 53–95

[22] Shalev-Shwartz, S., Ben-David, S.: '*Understanding machine learning: from theory to algorithms*' (Cambridge University Press New York, USA, 2014, 1st edn.)

[23] Neuhaus, S., Zimmermann, T., Holler, C., *et al.*: 'Predicting vulnerable software components'. Proc. 14th Conf. Computing Communication Security, Alexandria, Virginia, USA, November 2007, pp. 529–540

[24] Ott, J., Atchison, A., Harnack, P., *et al.*: 'A deep learning approach to identifying source code in images and video'. Proc. – Int. Conf. Software Engineering, Gothenburg, Sweden, May–June 2018, pp. 376–386

[25] 'Google Scholar'. Available at https://scholar.google.com/, accessed 15 January 2020

[26] 'IEEE Xplore'. Available at https://scholar.google.com/, accessed 15 January 2020

[27] 'ACM Digital Library'. Available at https://dl.acm.org/, accessed 15 January 2020

[28] 'Springer Link'. Available at https://link.springer.com/, accessed 15 January 2020

[29] 'Wiley Online Library'. Available at https://onlinelibrary.wiley.com/, accessed 15 January 2020

[30] 'Science Direct'. Available at https://www.sciencedirect.com/, accessed 15 January 2020

[31] Wohlin, C.: 'Guidelines for snowballing in systematic literature studies and a replication in software engineering'. Evaluation and Assessment in Software Engineering (EASE 2014), London, UK, May 2014, pp. 1–10

[32] Khan, M., Sherin, S., Iqbal, M., *et al.*: 'Landscaping systematic mapping studies in software engineering: A tertiary study', *J. Syst. Softw.*, 2019, **149**, (24), pp. 396–436

[33] Khan, M., Sherin, S., Iftikhar, S., *et al.*: 'Empirical studies omit reporting necessary details: A systematic literature review of reporting quality in model based testing', *Comput. Stand. Interfaces*, 2018, **55**, (25), pp. 156–170

[34] Zhou, Y., Zhang, H., Huang, X., *et al.*: 'Quality assessment of systematic reviews in software engineering: a tertiary study'. Proc. of the 19th Int. Conf. on Evaluation and Assessment in Software Engineering (EASE), Nanjing, China, April 2015, pp. 1–14

[35] Liu, B., Shi, L., Cai, Z., *et al.*: 'Software vulnerability discovery techniques: A survey'. Proc. – 2012 4th Int. Conf. Multimedia Security (MINES 2012), Nanjing, China, November 2012, pp. 152–156

[36] Shahriar, H., Zulkernine, M.: 'Mitigating program security vulnerabilities: approaches and challenges', *Comput. Surv.*, 2012, **44**, (3), pp. 1–46

[37] Li, Z.J., Shao, Y.: 'A survey of feature selection for vulnerability prediction using feature-based machine learning'. ACM Int. Conf. Proceeding Series, Tokyo, Japan, December 2019, pp. 36–42

[38] Malhotra, R.: 'A systematic review of machine learning techniques for software fault prediction', *Appl. Soft Comput.*, 2015, **27**, pp. 504–518

[39] Ghaffarian, S.M., Shahriari, H.R.: 'Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey', *ACM Comput. Surv.*, 2017, **50**, (4), pp. 1–36

[40] Jie, G., Xiao-Hui, K., Qiang, L.: 'Survey on software vulnerability analysis method based on machine learning'. Proc. IEEE 1st Int. Conf. Data Science Cyberspace (DSC), Changsha, China, February 2016, pp. 642–647

[41] Allamanis, M., Barr, E.T., Devanbu, P., *et al.*: 'A survey of machine learning for big code and naturalness', *ACM Comput. Surv.*, 2018, **51**, (4), pp. 1–36

[42] Lin, G., Wen, S., Han, Q.L., *et al.*: 'Software vulnerability detection using deep neural networks: a survey', *Proc. IEEE*, 2020, **108**, (10), pp. 1–24

[43] Sabottke, C., Suciu, O., Dumitras, T.: 'Vulnerability disclosure in the age of social media: exploiting twitter for predicting real-world exploits'. Proc. USENIX Security Symp., Washington, D.C., USA, August 2015, pp. 1041–1056

[44] Ramos, D.A., Engler, D.R.: 'Under-constrained symbolic execution: correctness checking for real code'. Proc. USENIX Security Symp., 2015, pp. 49–64

[45] 'CheckMarx Software Official Website'. Available at https://www.checkmarx.com, accessed 03 January 2020

[46] Cadar, C., Dunbar, D., Engler, D.R.: 'KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs'. OSDI'08: Proc. of the 8th USENIX Conf. on Operating systems design and implementation, San Diego, CA, USA, December 2008, pp. 209–224

[47] Pewny, J., Schuster, F., Bernhard, L., *et al.*: 'Leveraging semantic signatures for bug search in binary programs'. Proc. 30th Annu. Computer Security Applications Conf. (ACSAC), New Orleans, Louisiana, USA, December 2014, pp. 406–415

[48] Henzinger, T.A., Jhala, R., Majumdar, R., *et al.*: 'Software verification with BLAST'. Proc. of the Int. SPIN Workshop on Model Checking of Software, Portland, OR, USA, May 2003, pp. 235–239

[49] Portokalidis, G., Slowinska, A., Bos, H.: 'Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation', *ACM SIGOPS Oper. Syst. Rev.*, 2006, **40**, (4), pp. 15–27

[50] Scandariato, R., Walden, J., Hovsepyan, A., *et al.*: 'Predicting vulnerable software components via text mining', *IEEE Trans. Softw. Eng.*, 2014, **40**, (10), pp. 993–1006

[51] Pang, Y., Xue, X., Namin, A. S.: 'Predicting vulnerable software components through N-gram analysis and statistical feature selection'. Proc. IEEE 14th Int. Conf. Machine Learning Application (ICMLA), 2015, pp. 543–548

[52] White, M., Vendome, C., Linares-Vasquez, M., *et al.*: 'Toward deep learning software repositories'. Proc. IEEE/ACM 12th Working Conf. Mining Software Repositories, 2015, pp. 334–345

[53] McCabe, T.J.: 'A complexity measure', *IEEE Trans. Softw. Eng.*, 1976, **SE-2**, (4), pp. 308–320

[54] Nagappan, N., Ball, T.: 'Use of relative code churn measures to predict system defect density'. Proc. 27th Int. Conf. Software Engineering (ICSE), 2005, pp. 284–292

[55] Engler, D., Chen, D. Y., Hallem, S., *et al.*: 'Bugs as deviant behavior: A general approach to inferring errors in systems code', *SIGOPS Oper. Syst. Rev.*, 2001, **35**, (5), pp. 57–72

[56] Li, Z., Zhou, Y.: 'PR-miner: automatically extracting implicit programming rules and detecting violations in large software code', *ACM SIGSOFT Softw. Eng. Notes*, 2005, **30**, (5), pp. 306–315

[57] Acharya, M., Xie, T., Pei, J., *et al.*: 'Mining API patterns as partial orders from source code: from usage scenarios to specifications'. Proc. 6th Joint Meeting European Software Engineering Conf. ACM SIGSOFT Symp. Foundations of Software Engineering, 2007, pp. 25–34

[58] Krizhevsky, A., Sutskever, I., Hinton, G.E.: 'Imagenet classification with deep convolutional neural networks', *Proc. Adv. Neural Inf. Process. Syst.*, 2012, **1**, pp. 1097–1105

[59] Bahdanau, D., Cho, K.H., Bengio, Y.: 'Neural machine translation by jointly learning to align and translate'. 3rd Int. Conf. Learning Representation (ICLR 2015) – Conf. Track Proc., 2015, pp. 1–15

[60] Yao, W., Cai, W., Wei, P.: 'A deep learning approach for detecting malicious JavaScript code', *Secur. Commun. Netw.*, 2016, **9**, pp. 1520–1534

[61] Ivergence, D., Montague, P., Vel, O.De: 'Maximal divergence sequential autoencoder for binary software vulnerability detection', *Am. Agric. Econ. Assoc.*, 2019, **86**, pp. 321–331

[62] Kim, Y.: 'Convolutional neural networks for sentence classification'. EMNLP 2014 – 2014 Conf. Empirical Methods National Language Processing Proc. Conf., 2014, pp. 1746–1751

[63] Ramsundar, B., Zadeh, R.B.: '*Tensorflow for deep learning: from linear regression to reinforcement learning*' (O'Reilly Media, Newton, MA, USA, 2018, 1st edn.)

[64] Yih, W.T., He, X., Meek, C.: 'Semantic parsing for single-relation question answering'. Proc. 52nd Annual Meeting Association Computing Linguistics, 2014, pp. 643–648

[65] Hochreiter, S., Schmidhuber, J.: 'Long short-term memory', *Neural Comput.*, 1997, **9**, (8), pp. 1735–1780

[66] Cho, K., Van Merriënboer, B., Gulcehre, C., *et al.*: 'Learning phrase representations using RNN encoder-decoder for statistical machine translation'. EMNLP 2014 – 2014 Conf. Empirical Methods National Language Processing Proc. Conf., 2014, pp. 1724–1734

[67] Dam, H.K., Tran, T., Pham, T., *et al.*: 'Automatic feature learning for vulnerability prediction', *IEEE Trans. Software Eng.*, 2017, **2017**, (1), p. 1

[68] Pang, Y., Xue, X., Wang, H.: 'Predicting vulnerable software components through deep neural network'. Int. Conf. on Deep Learning Technologies 2017, 2017, pp. 6–10

[69] Wang, Y., Wu, Z., Wei, Q., *et al.*: 'Neufuzz: efficient fuzzing with deep neural network', *IEEE Access*, 2019, **7**, pp. 36340–36352

[70] Wu, F., Wang, J., Liu, J., *et al.*: 'Vulnerability detection with deep learning'. 2017 3rd IEEE Int. Conf. on Computer and Communications Vulnerability, 2017, pp. 1298–1302

[71] Xiaomeng, W., Tao, Z., Runpu, W., *et al.*: 'CPGVA: code property graph-based vulnerability analysis by deep learning'. 2018 10th Int. Conf. on Advanced Infocomm Technology (ICAIT 2018), 2018, pp. 184–188

[72] Xu, A., Dai, T., Chen, H., *et al.*: 'Vulnerability detection for source code using contextual LSTM'. 2018 5th Int. Conf. on Systems and Informatics (ICSAI 2018), 2018, pp. 1225–1230

[73] Lin, G., Zhang, J., Luo, W., *et al.*: 'Software vulnerability discovery via learning multi-domain knowledge bases', *IEEE Trans. Dependable Secur. Comput.*, 2019, **5971**, (c), pp. 1–1

[74] Nguyen, V., Le, T., Le, T., *et al.*: 'Deep domain adaptation for vulnerable code function identification'. Proc. of the Int. Joint Conf. on Neural Networks, 2019, pp. 4–11

[75] Saccente, N., Dehlinger, J., Deng, L., *et al.*: 'Project achilles: A prototype tool for static method-level vulnerability detection of Java source code using a recurrent neural network'. Proc. – 2019 34th IEEE/ACM Int. Conf. Automatic Software Engineering Workshop (ASEW 2019), 2019, pp. 114–121

[76] Peng, H., Mou, L., Li, G., *et al.*: 'Building program vector representations for deep learning'. Proc. Int. Conf. Knowledge Science, Engineering Manage, 2015, pp. 547–553

[77] Zou, D., Wang, S., Xu, S., *et al.*: 'µ VulDeePecker' a deep learning-based system for multiclass vulnerability detection', *IEEE Trans. Dependable Secur. Comput.*, 2019, pp. 1–13

[78] Choi, M.J., Jeong, S., Oh, H., *et al.*: 'End-to-end prediction of buffer overruns from raw source code via neural memory networks'. IJCAI Int. Joint Conf. on Artificial Intelligence, 2017, pp. 1546–1553

[79] Dormuth, J., Gelman, B., Moore, J., *et al.*: 'Logical segmentation of source code'. Proc. of the 31st Int. Conf. on Software Engineering and Knowledge Engineering', 2019, pp. 717–722

[80] Duan, X., Wu, J., Ji, S., *et al.*: 'Vulsniper: focus your attention to shoot fine-grained vulnerabilities'. IJCAI Int. Joint Conf. on Artificial Intelligence, 2019, pp. 4665–4671

[81] Harer, J.A., Kim, L.Y., Russell, R.L., *et al.*: 'Automated software vulnerability detection with machine learning', 2018, arXiv:1803.04497. [Online]. Available: http://arxiv.org/abs/1803.04497

[82] Li, R., Feng, C., Zhang, X., *et al.*: 'A lightweight assisted vulnerability discovery method using deep neural networks', *IEEE Access*, 2019, **7**, pp. 80079–80092

[83] Li, Z., Zou, D., Tang, J., *et al.*: 'A comparative study of deep learning-based vulnerability detection system', *IEEE Access*, 2019, **7**, pp. 103184–103197

[84] Li, Z., Zou, D., Xu, S., *et al.*: 'SySeVR: a framework for using deep learning to detect software vulnerabilities', 2018, pp. 1–13

[85] Li, Z., Zou, D., Xu, S., *et al.*: 'Vuldeepecker: A deep learning-based system for vulnerability detection'. Network and Distributed Systems Security (NDSS) Symp. 2018', 2018, pp. 1–15

[86] Russell, R., Kim, L., Hamilton, L., *et al.*: 'Automated vulnerability detection in source code using deep representation learning'. Proc. – 17th IEEE Int. Conf. on Machine Learning and Applications (ICMLA 2018), 2018, pp. 757–762

[87] Zhou, Y., Liu, S., Siow, J., *et al.*: 'Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks', 2019, pp. 1–11

[88] Zou, Y., Kontogiannis, K.: 'Towards A portable XML-based source code representation', *Comput. Eng.*, 2001, **343**, p. 353

[89] Jones, J.: 'Abstract syntax tree implementation idioms', *Pattern Lang. Progr.*, 2003, pp. 1–10

[90] Schmidhuber, J.: 'Deep learning in neural networks: an overview', *Neural Netw.*, 2014, **61**, pp. 85–117