


Software Source Code Vulnerability Static Analysis Based on Machine Learning: A Survey

Gaigai Tang ^{1,†,‡} , Firstname Lastname ^{2,†} and Firstname Lastname ^{2,*}

¹ Affiliation 1; e-mail@e-mail.com

² Affiliation 2; e-mail@e-mail.com

* Correspondence: e-mail@e-mail.com; Tel.: (optional; include country code; if there are multiple corresponding authors, add author initials) +xx-xxxx-xxx-xxxx (F.L.)

† Current address: Affiliation 3.

‡ These authors contributed equally to this work.

Abstract: A single paragraph of about 200 words maximum. For research articles, abstracts should give a pertinent overview of the work. We strongly encourage authors to use the following style of structured abstracts, but without headings: Background: place the question addressed in a broad context and highlight the purpose of the study; Methods: describe briefly the main methods or treatments applied; Results: summarize the article's main findings; Conclusions: indicate the main conclusions or interpretations. The abstract should be an objective representation of the article, it must not contain results which are not presented and substantiated in the main text and should not exaggerate the main conclusions.

Keywords: keyword 1; keyword 2; keyword 3

1. Introduction

Vulnerability is a defect in system design, specific implementation, system configuration or security strategy of hardware, software and protocols of computer systems. Once the defect is found and maliciously exploited, it will make the attacker access or destroy the system without authorization, thus affecting the normal operation of the computer system and even causing security damage. Currently, there are 189,414 records reported by Common vulnerabilities and Exposures (CVE)[1], and more than 20,000 records reported by National Vulnerability Database (NVD)[2] within this year. The number of vulnerabilities is always increasing, which brings continuous pressure to the software security industry. It is essential to ensure software security from the source.

At present, the most commonly used method is code audit. The purpose of code security audit is to identify and repair various potential risks that may affect the system security by reviewing the source code, and to reduce the system risk by improving the quality of the code itself. Due to the large amount of code, code security audit generally uses static analysis tools to automatically detect code and generate code security audit reports. The static analysis methods are mainly include vulnerability detection and repair methods. With the introduction of artificial intelligence technology, the existing vulnerability detection and repair methods have gradually realized intelligence, effectively improving the practical performance. The AI technology introduced mainly includes machine learning and deep learning, especially natural language processing (NLP), graph neural network (GNN), and so on.

This survey investigates the software source code vulnerabilities static analysis methods based on machine learning. We discuss the differences and connections between existing related methods, classifies them according to their main characteristics, and points out the limitations of the methods. Finally, the general status and limitations of static analysis technology based on machine learning are summarized and analyzed, and some open

Citation: Lastname, F.; Lastname, F.; Lastname, F. Software Source Code Vulnerability Static Analysis Based on Machine Learning: A Survey. *Systems* **2022**, *1*, 0. <https://doi.org/>

Received:

Accepted:

Published:

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Submitted to *Systems* for possible open access publication under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

problems are proposed.

To summarize the work of this survey, our key contributions are shown as follows. Firstly, We have investigated the latest research progress in the past five years on software source code vulnerabilities static analysis methods based on machine learning. Table 1 gives an overview of the key technologies of these methods. Secondly, we classify the existing vulnerability detection and repair methods according to multiple features, and analyze the advantages and disadvantages of methods in each classification. Finally, we discuss the limitations and challenges of machine learning based static analysis methods, and propose relevant open issues and some suggestions.

2. Related Literature Reviews

With the rapid development of machine learning technology, the static analysis technology based on machine learning is also updated rapidly and iteratively. We first investigated and analyzed the relevant reviews in recent five years, mainly analyzed their classification forms of existing methods, and explained the novelty of our work.

2.0.1. Software vulnerability detection using machine learning

Seyed Mohammad Ghaffarian and et.al [3] reviewed the works in the field of software vulnerability analysis and discovery that utilize machine-learning and data mining techniques. They categorize the reviewed works in four main categories, which in summary are defined and discriminated as follows: vulnerability prediction models based on software metrics, which use machine-learning approach to build a prediction model based on well-known software metrics as the feature set; anomaly detection approaches, which utilize an unsupervised learning approach to automatically extract a model of normality or mine rules from the software source code; vulnerable code pattern recognition, which utilize a (mostly supervised) machine-learning approach to extract patterns of vulnerable code segments from many vulnerability code samples; miscellaneous approaches, which utilized techniques from the fields of AI and data science for software vulnerability analysis and discovery.

Jian Jiang and et.al [4] gave a brief study of the software vulnerability discovery using machine learning techniques, who divided the vulnerability discovery techniques into static analysis, symbolic execution and fuzzing. The static analysis can be classified as source code analysis and binary analysis based on targets' type. For source code analysis, the authors pointed out that introducing reinforcement learning to guide the learning of the training model and use transfer learning to address the problem of training data and test case vary significantly.

Abubakar Omari Abdallah Semasaba and et.al [5] provided 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). The taxonomy of deep learning techniques for source code vulnerability analysis is: according to vulnerability analysis objectives, there are binary class vulnerability detection, multiclass vulnerability detection, fine-grained vulnerability detection; according to vulnerability analysis features, there are feature representation based (such as graph-based feature representation, code block-based feature representation, text-based feature representation, mixed features-based representation), portable output features (such as Abstract syntax trees, code gadget, Code property graphs, lexed representation of source code); according to DL algorithms, there unsupervised pre-trained networks, convolutional neural networks, recurrent neural networks. For issues and challenges, the authors pointed out having a data set that consists of multiple programming languages would require further exploration; the

focus on libraries/API function calls have been documented as a limitation; related works used a variety of data sets that represent both vulnerable and non-vulnerable functions.

Guanjun Lin and et.al [6] reviewed the current literature adopting deep learning/neural network-based approaches for detecting software vulnerabilities, aiming at investigating how the state-of-the-art research leverages neural techniques for learning and understanding code semantics to facilitate vulnerability discovery. The authors categorized the reviewed studies into four categories of feature representations : the graph-based feature representations that apply DNNs for learning feature representations from different types of graph-based program representations, including the AST, 2 the CFG, the PDG, and a combination of them; the sequence-based feature representations utilize DNNs for extracting feature representations from sequential code entities, such as the execution trace, the function call sequences, and the variable flow/sequence; the text-based feature representations that the feature representations are directly learned from the surface text of the code; the mixed feature representations , including recent studies that combined the above mentioned three types of feature representations. For the challenges and future works, the authors pointed out the urgent need for a standard benchmarking data set to act as a unified metric for evaluating and comparing the effectiveness of proposed approaches; expertise and knowledge from software engineering and cybersecurity fields are still in need of defining representative and effective features to guide the design of the network architecture for better discovering vulnerable semantics and patterns which are latent and complex; to fill the semantic gap by enabling a neural model to better reason with the semantics of programming languages; a more effective embedding technique providing a learning system with richer semantic information to learn from is desirable; the attention mechanism can serve as a means for explaining the neural network model.

Wang, Jingjing and et.al[7] systematically summarized the development trends and different technical characteristics in this field from the perspectives of the intermediate representation of source code and vulnerability prediction model. The source code representation in vulnerability analysis using machine learning techniques are categorized to code metrics and word Characteristics; code properties, including Abstract Syntax Tree (AST), Control Flow Graph (CFG), Program Dependency Graph (PDG) and so on; code semantics, including semantic features of source code in the form of text. The vulnerability prediction models using machine learning are divided into conventional machine learning-based (SVM, Decision Trees and et.al) and Deep learning-based (CNN,RNN). Moreover, the authors summarized the feasible future research directions, dealing with the shortage of datasets with ground truth, mitigation of class imbalance,improvement of model interpretability.

Hazim Hanif and et.al [8] presented the research interests' taxonomy in software vulnerability detection, such as methods, detection, features, code and dataset. Where methods include manual analysis and conventional techniques, detection includes detection performance and detection class, features include classic metrics and features, overfitting. Moreover, this study selects only the most common issues which are gaining traction and attention in software vulnerability detection. The issues are dataset, multi-vulnerabilities detection, transfer learning and real-world application.

Senanayake, Janaka and et.al [9] focuses on Android application analysis and source code vulnerability detection methods and tools. It highlights the advantages, disadvantages, applicability of the proposed techniques and potential improvements of those studies. Several methods including machine learning, deep learning, heuristic-based methods, and formal methods can be applied to detect source code vulnerabilities with the use of static analysis, dynamic analysis and hybrid analysis. Further research should also be conducted to identify the possible ways of integrating the detection methods into Android app devel-

opment environments as tools or plugins. Moreover, further research can be conducted to integrate the explainable AI techniques with Android source code vulnerability detection mechanisms.

Pooja, S and et.al [10] made a qualitative comparative analysis on ML, DL, GNN based model to Vulnerability detection model (VDM), thus addressing the challenges to choose suitable architecture, feature representation, datasets, and preprocessing required for designing a VDM. The different feature representation techniques used in AI based VDM are textual token, code segment, and graph. And the varying granularity levels are token, code gadget, function, program, file, release.

Marjanov, Tina and et.al[11] reviewed machine learning approaches for detecting (and correcting) vulnerabilities in source code. They divided the related studies primary by the detect type (semantic, syntactic, vulnerability), representation type (token, token, graph), methods (RNN, CNN,GNN), language (C, Python, Java, JavaScript), and the dataset used. Furthermore, the authors pointed the challenges that are expand detection tools for all defect types, advanced (semantic) representations and embeddings, expand to more languages, formalize goals and metrics for tools and simplify output for developers, collect, standardize high-quality, realistic, and representative datasets across all defect types and language.

2.0.2. Software vulnerability repair using machine learning

Claire Le Goues and rt.al [12] surveyed the automatic program repair methods, they classified them to two main approaches: heuristic repair and constraint-based repair. These techniques can sometimes be enhanced by machine learning, which can be called learning-aided repair. Heuristic repair includes heuristic search methods, which employ a generate-and-test methodology, constructing and iterating over a search space of syntactic program modifications. Constraint-based repair, In contrast to heuristic repair techniques, constraint-based techniques proceed by constructing a repair constraint that the patched code should satisfy. Learning-based repair falls approximately into three categories that vary by the extent to which they exploit learning during the repair process. One line of work learns from a corpus of code a model of correct code, which indicates how likely a given piece of code is with regard to the code corpus. The approach then uses this model to rank a set of candidate patches to suggest the most realistic patches first. Another line of work infers code transformation templates from successful patches in commit histories that summarize how patches modify buggy code into correct code. These transformation templates can then be used to generate repair candidates. The third line of work not only improves some part of the repair process through learning, but also trains models for end-to-end repair. Further, the authors pointed out challenges, that are, the quality challenge is about increasing the chance an automatically identified repair provides a correct fix that is easy to maintain in the long term; the scope challenge is about further extending the kinds of bugs and programs to which automated repair applies; the final challenge is about integrating repair tools into the development process.

Luca Gazzola and et.al [13] organized the knowledge in the automatic software repair, illustrating the algorithms and the approaches, comparing them on representative examples. They divided the repairing process to three steps. The localization step identifies the locations where a fix could be applied to (note that the faulty statements are not always the only good locations for fixes). The patch/fix step generates fixes that modify the software in the code locations returned by the localization step. The verification step checks if the synthesized fix has actually repaired the software. The two main classes of fix generated approaches depending on the way repair is defined and addressed: generate-and-validate

and semantics-driven approaches. This review did not investigate repair methods using machine learning.

Zhidong Shen and et.al [14] categorized the patching technology to grammar-based and semantic-based technologies. The grammatical error-based program repair technique modifies the error code by learning the code grammar features to achieve certain language specifications. The semantic error-based program repair technique generally refers to modifying the program code so that the actual program behaviour is substantially consistent with the behaviour expected by the programmer.

Table 1. Summary of main aspects of relevant survey papers

Aspect	Item	Related Papers											
		detection						repair					
		[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
category indicator	code metrics					✓	✓						✓
	dataset						✓			✓			
	features			✓	✓		✓			✓			✓
	objectives	✓		✓				✓		✓	✓	✓	✓
	target type	✓	✓							✓			
	technique /method	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓
open issue /opportunity	dataset			✓	✓	✓	✓		✓	✓		✓	✓
	feature extraction				✓				✓	✓		✓	
	granularity			✓					✓				
	Interpretability				✓	✓		✓				✓	✓
	multiple languages			✓						✓	✓		
	multiple categories						✓			✓	✓		
	new technique	✓					✓				✓	✓	✓

Table 1 gives a summary of the main aspects of relevant articles, such as category indicator and open issue/opportunity. These papers are arranged from 2017 to 2022 according to the publication time. We can see that new papers cover more items than earlier papers.

3. Software Source code vulnerability static detection based on machine learning

Machine learning based source code static vulnerability detection methods can be divided into two categories: similarity based methods and pattern based methods. The main idea of similarity based methods is that two pieces of similar code may have the same vulnerability. The main idea of

3.1. similarity-based

Similarity-based approaches first abstractly represents the source code, and then obtains the similarity results of the two pieces of source code through similarity comparison method, and then determines whether there are vulnerabilities to implement vulnerability detection according to user-defined rules. According to the representation form of source code, we divide these approaches into sequence based representation and graph based representation approaches.

3.1.1. txt-based approaches

3.1.2. seq-based approaches

Zhen Li and et.al[15] proposed VulPecker, The key insight underlying VulPecker is to leverage (i) a set of features that we define to characterize patches, and (ii) code-similarity

algorithms that have been proposed for various purposes, while noting that no single code-similarity algorithm is effective for all kinds of vulnerabilities. Experiments show that VulPecker detects 40 vulnerabilities that are not published in the National Vulnerability Database (NVD). it also built a Vulnerability Patch Database (VPD) and a Vulnerability Code Instance Database (VCID), which correspond to the C/C open source products that have some vulnerabilities according to the NVD. designing algorithms to automatically select the code-similarity algorithm(s) that is effective for one specific vulnerability, allowing us to systematically evaluate which code-similarity algorithms are effective for which vulnerabilities.

Junaid Akram and et.al [16] proposed VCIPR, using a fast, token-based approach to detect vulnerabilities at function level granularity. This approach is language independent, which supports multiple programming languages including Java, C/C++, JavaScript. VCIPR detects most common repair patterns in patch files for the vulnerability code evaluation. We build fingerprint index of top critical CVE's source code, which were retrieved from a reliable source. Then we detect unpatched (vulnerable/non-vulnerable) code fragments in common open source software with high accuracy. A comparison with the state-of-the-art tools proves the effectiveness, efficiency and scalability of our approach.

Junaid Akram and et.al [17] proposed a vulnerability detection technique to detect vulnerabilities in software, as well as shared libraries at source code level. We crawl the vulnerable source code by tracing and locating the patch files from different web sources according to their CVE-numbers and built a fingerprint index of 2931 vulnerable files. Then we developed a vulnerability detection approach based on code clone detection technique and detect hundreds of vulnerabilities in thousands of GitHub open source projects, which are not noticed before as vulnerable. We detected vulnerabilities in some very famous recently available software, including latest version of Linux, HTC-kernel, FindX-8.1-kernel, and in 7-TB of C/C++ source code (152,823 open source projects). It proposed a scalable vulnerability detection approach to detect vulnerabilities in different software at source code level. We extracted different features from the source code before creating an index of fingerprint, then we detected vulnerabilities in the subject system by applying feature matching filters on it. This approach is implemented practically and the experiments shows the effectiveness and scalability of our approach.

Hao Sun and et.al [18] presented a metric learning-based approach, which trains a code similarity detector on a data set of vulnerabilities and patches. In particular, we pay our efforts in two directions, a data set that characterizes vulnerabilities and a metric learning model that learns similarity in the view of vulnerability. First, we prepare a data set of CVE bunches, 2 each of which contains multiple vulnerable functions and patched functions associated with one CVE, as illustrated in Fig. 1. These functions for each CVE can be acquired from various versions of the affected software program, and they follow two rules. a) For two vulnerable functions of the same CVE across two program versions, the vulnerability snippet will keep despite of the code change. b) For one vulnerable function and the associated patched function, the vulnerability snippet will disappear no matter how the code changes. Therefore, each bunch of functions potentially provide vulnerability characteristics of one CVE. Second, in the view of vulnerability, two vulnerable functions across different versions should be treated as similar, even they experience code changes. On the other hand, a vulnerable function and its patched function, even looks similar in syntax, should be treated as different since the vulnerability snippet has been removed. Inspired by this, we employ the Siamese model to learn the similarity between two vulnerable functions, while learning the difference between vulnerable function and patched function. In addition, we incorporate BiLSTM and Attention network in the Siamese model, which helps generate more accurate and focused representations of functions for detection. In this way, the trained model can perceive similarity in the view of vulnerability instead of

merely semantics and syntax. Linux CVEs, which helps characterize vulnerability snippets. The data set is available in Github. 3 • Second, a detection model using Siamese network cooperated with BiLSTM and Attention network. By taking pairs of vulnerable functions and patched functions, the model is able compute the similarity between two functions, so as to detect vulnerabilities. We plan to release the source code upon the publication of this paper. • Third, the system implementation and evaluation on the data set to prove the effectiveness of the proposed model. the proposed model (VDSimilar) achieves about 97.17% in AUC value of OpenSSL (where the Attention network contributes 1.21% than BiLSTM in Siamese), which is more outstanding than the most advanced methods based on deep learning.

Solmaz Salimi and et.al [19] extracted slices that are considered to be related to vulnerabilities from known vulnerable programs and then abstract the statements in the slice by eliminating local naming conventions used in variables and function names. Similarly we decompose the target program into abstract program slices. Now if the target program contains vulnerabilities similar to the vulnerable programs we have used to generate VRSs, then we will be able to identify those vulnerabilities. We introduce a method which, by leveraging program slicing (Weiser and slicing, 1981), is capable of extracting succinct and self-contained vulnerability-relevant statements, VRS, from known vulnerable programs. • We propose an efficient search technique, where by slicing the target program for all possible slicing criteria and then comparing a given VRS with the set of target program slices, can decide on the existence of vulnerabilities in the target program. • To evaluate our approach, we developed a system called VulSlicer with two primary goals: (i) to extract VRSs from known vulnerable programs, and (ii) to search for vulnerabilities within target programs by decomposing the program and preparing comparable code segments with VRSs. • We have been able to identify several new vulnerabilities, 3 of which have been assigned a CVE number.

3.1.3. graph-based approaches

Fabian Yamaguchi and et.al[20] proposed a method for assisting a security analyst during auditing of source code. Instead of striving for an automated solution, we aim at rendering manual auditing more effective by guiding the search for vulnerabilities. Based on the idea of vulnerability extrapolation [33], our method proceeds by extracting abstract syntax trees from the source code and determining structural patterns in these trees, such that each function in the code can be described as a mixture of the extracted patterns. The patterns contain subtrees with nodes corresponding to types, functions and syntactical constructs of the code base. This representation enables our method to decompose a known vulnerability and to suggest code with similar properties—potentially suffering from the same flaw—to the analyst for auditing. We evaluate the efficacy of our method using the source code of four popular open-source projects: LibTIFF, FFmpeg, Pidgin and Asterisk. We first demonstrate in an quantitative evaluation how functions are decomposed into structural patterns and how similar code can be identified automatically. In a controlled experiment we are able to narrow the search for a given vulnerability to 8.7% of the code base and consistently outperform non-structured approaches for vulnerability extrapolation. We also study the discovery of real vulnerabilities in a qualitative evaluation, where we are able to discover 10 zero-day vulnerabilities in the source code of the four open-source projects.

Xin Zhang and et.al [21] in order to restore the syntax and semantics of the source code as much as possible and improve the detection efficiency, we construct a weight graph model VDBWGD. Firstly, VDBWGD automatically collects vulnerability samples, accurately obtains the location of vulnerability function through some methods, and extracts vulnerability characteristic keywords from it. Then we divide vulnerabilities according to sensitive points and get a high-quality vulnerability data set through code variation rules.

To deal with these data better, we use static analysis tools to transform them into graph representation structure and weight the graph according to the distance of sensitive words. Then, we transform the extracted code block syntax semantic information into vectors through a deep learning model. Through these methods, we can extract the deep structure information of the function, keep the complete syntax and semantic information, and remove redundant information. F1 and accuracy have reached the best compared with the four state-of-the-art models. contributions are summarized as follows: • A code variation method is designed according to the vulnerability triggering logic and the programmer's programming style. It generates a lot of relatively valuable vulnerability data. • The weight graph model. It pays more attention to the feature points of the vulnerability function. • This model's code block vectorization method can keep the rich syntax and semantic information of functions and simplify the complex graphic information.

3.2. *pattern-based*

pattern based method is to learn the generation pattern of some vulnerabilities through prior knowledge, and then implement vulnerability detection by pattern matching the input source code. This kind of methods mainly include the representation method of source code and pattern learning model.

3.2.1. *txt-based*

Limin Yang and et.al [22] proposed VulDigger, constructing a large-scale vulnerability-contributing changes (VCCs) dataset in a semiautomatic fashion, then it manifest a classification tool with a mixture of established and new metrics (e.g., the maximum changes has been made in the past for files modified in a change) derived from vulnerability prediction. The precision of such tool is extremely promising (i.e., 92=%) for a cost-aware software team. further develop a regression model to locate most skeptical changes with fewer lines to inspect. Such model is capable of pinpointing 31% of all VCCs with 20% of the effort it would take to audit all changes. Finally, we improve the algorithm of mapping vulnerabilities to VCCs by placing more constraints to remove false alarms and then construct a large-scale dataset for Mozilla Firefox.

Marian Gawron and et.al [23] proposed an automatic approach to classify vulnerabilities and their natural language descriptions into those formal attributes. Therefore, the first goal was to investigate the feasibility of an automated approach and secondly evaluate the accuracy. Thus, we implemented two different approaches that are capable of an automated classification. We used Naive Bayes [12] as one approach, since it is a widely distributed method to solve classification problems. For the other approach we rely on Neural Networks, as modern natural language processing systems utilize Neural Networks as well. At first, we identified the most important characteristics of vulnerabilities, namely the CVSS attributes availability, integrity, confidentiality, and attack range. Then we created our training, testing, and validation data sets and trained our model on the vulnerability description. Both approaches required some additional preprocessing steps, as it was already described in Sect.3. The application of the Naive Bayes classification has the advantage that it could directly work with natural language, whereas the Neural Network required one additional transformation step. After the training we evaluated the two approaches on the test data set. In addition to the test data set we also created one validation data set, since the Neural Network approach also uses the test data to train the model or adjust the parameters. So the most important evaluation metric is the accuracy of the validation data set, since it was not used during the training procedure. We found that the automated classification is possible and the accuracy for single attributes, which was around 70% to 90% depending on the attribute and the data set, is also satisfying. The important metric of the combined attributes results in lower accuracy, but it was still possible to achieve an accuracy of 60% to 70% on the test data and around 50% on the validation data.

Boris Chernis and et.al [24] proposed a methodology for analyzing features from C source code to classify functions as vulnerable or non-vulnerable. After finding 100 programs on GitHub, we parsed out all functions from these programs. We then extracted trivial features (function length, nesting depth, string entropy, etc) and non-trivial features (n-grams and suffix trees) from these functions. The statistics for these features were arranged in a table, which was split into training data and test data. Several different classifiers, including Naive Bayes, k nearest neighbors, k means, neural network, support vector machine, decision tree, and random forest, were used to classify the test samples. The trivial features produced the best classification result, with an accuracy of 75%, while the best n-grams result was 69% and the best suffix trees result was 60%. Relatively simple features (character count, character diversity, entropy, maximum nesting depth, arrow count, 'if' count, 'if' complexity, 'while' count, and 'for' count) were extracted from these functions, and so were complex features (character n-grams, word n-grams, and suffix trees). The simple features performed unexpectedly better compared to the complex features (74% accuracy compared to 69% accuracy).

Xiaoning Du and et.al [25] proposed and implemented a generic, lightweight and extensible framework, LEOPARD, to identify potentially vulnerable functions through program metrics. LEOPARD requires no prior knowledge about known vulnerabilities. It has two steps by combining two sets of systematically derived metrics. First, it uses complexity metrics to group the functions in a target application into a set of bins. Then, it uses vulnerability metrics to rank the functions in each bin and identifies the top ones as potentially vulnerable. i.e., complexity metrics and vulnerability metrics. Complexity metrics capture the complexity of a function in two complementary dimensions: the cyclomatic complexity of the function, and the loop structures in the function. Vulnerability metrics reflect the vulnerable characteristics of functions in three dimensions: the dependency of the function, pointer usage in the function, and the dependency among control structures within the function. Our experimental results on 11 real-world projects have demonstrated that, LEOPARD can cover 74.0% of vulnerable functions by identifying 20% of functions as vulnerable and outperform machine learning-based and static analysis-based techniques.

Canan Batur Sahin and et.al [26] proposed an improved feature selection method to reveal optimal feature sets by enhancing the classification accuracy of static analysis. This paper's proposed method is the immune-based feature selection that is utilized to discover optimal feature sets by improving the classification accuracy of static analysis. An immune feature selection method is proposed as a potential solution in the fixed analysis domain. Experiments are conducted on the three standard datasets. The detailed assessment of the impacts of the proposed artificial immune system dynamics on the prediction of software vulnerabilities, the design of a novel feature selection algorithm for an SVP problem, and the assessment of seven classification algorithms on three public datasets constitute the contributions of this article. The proposed method improved the current dataset generation framework to involve additional predictive software features. So, a realworld static analysis dataset is used based on three opensource PHP applications. The proposed feature selection method is presented and tested to determine the benefit rate of choosing relevant features and to study the influence on the classification of vulnerable components. An in-depth investigation of the proposed feature selection method to enhance vulnerability classification is provided. Compared to conventional immune-based algorithms, finding a better feature subset for classification with immune algorithmbased feature selection is possible. The obtained results showed that the proposed feature selection method achieved better results compared to other comparative methods. Moreover, the proposed method improved the usefulness of static analysis tools by learning optimized historical data features.

3.2.2. seq-based

Bo Shuai and et.al [27] proposed an intelligent approach based on Latent Dirichlet Allocation (LDA) model and SVM. LDA model is introduced into feature selection and integrate the location information into the feature words. The Weighted Location LDA (WL-LDA) model could establish the generative model bases on weighted location themes which is better than on feature words. Then the multiclass SVM classifier is constructed using Huffman Tree arithmetic which could make good use of the prior knowledge of distribution of the number of vulnerabilities. The Huffman Tree SVM (HTSVM) is trained based on the implicit topics-words matrix, which is obtained from the feature words probabilistic distributions of WL-LDA model. The approach combines the useful features of vulnerabilities, the outstanding feature dimensionality reduction and text representation capabilities of LDA with the powerful classification ability of SVM to improve the vulnerabilities classification performance.

Henning Perl and et.al [28] proposed VCCFinder, a classifier that can identify potentially vulnerable commits with a significantly lower false positive rate while retain high recall rates. Therefore, unlike most existing tools for vulnerability finding, we don't focus solely on code metrics, but also leverage the rich metadata contained in code repositories. To evaluate the effectiveness of our approach, we conduct a large-scale evaluation of 66 GitHub projects with 170,860 commits, gathering both metadata about the commits as well as mapping CVEs to commits to create a database of vulnerability-contributing commits (VCCs) and a benchmark for future research. We conducted a statistical analysis of these VCCs and trained a Support Vector Machine (SVM) to detect them based on the combination of code metric analysis and GitHub metadata.

Jacob A. Harer and et.al [29] presented a data-driven approach to vulnerability detection using machine learning, specifically applied to C and C programs. We first compile a large dataset of hundreds of thousands of open-source functions labeled with the outputs of a static analyzer. We then compare methods applied directly to source code with methods applied to artifacts extracted from the build process, finding that source-based models perform better. We also compare the application of deep neural network models with more traditional models such as random forests and find the best performance comes from combining features learned by deep models with tree-based models.

Rebecca L. Russell and et.al [30] leveraged the wealth of C and C open-source code available to develop a largescale function-level vulnerability detection system using machine learning. To supplement existing labeled vulnerability datasets, we compiled a vast dataset of millions of open-source functions and labeled it with carefully-selected findings from three different static analyzers that indicate potential exploits. Using these datasets, we developed a fast and scalable vulnerability detection tool based on deep feature representation learning that directly interprets lexed source code. Achieving the best overall results using features learned via convolutional neural network and classified with an ensemble tree algorithm.

Zhen Li and et.al [31] propose the first systematic framework for using deep learning to detect vulnerabilities. It introduce and define the notions of Syntaxbased Vulnerability Candidates (SyVCs) and Semantics-based Vulnerability Candidates (SeVCs), and design algorithms for computing them. Intuitively, SyVCs reflect vulnerability syntax characteristics, and SeVCs extend SyVCs to accommodate the semantic information induced by data dependency and control dependency. Correspondingly, the framework is called Syntax-based, Semantics-based, and Vector Representations, or SySeVR for short. It produces a dataset of 126 types of vulnerabilities caused by various reasons from the National Vulnerability Database (NVD) and the Software Assurance Reference Dataset (SARD), Moreover, we

detected 15 vulnerabilities that were not reported in the NVD.

Zhen Li and et.al [32] proposed VulDeePecker, using code gadgets to represent programs and then transform them into vectors, where a code gadget is a number of (not necessarily consecutive) lines of code that are semantically related to each other. It is the first deep learning-based vulnerability detection system, which aims to relieve human experts from the tedious and subjective work of manually defining features and reduce the false negatives that are incurred by other vulnerability detection systems. We have collected, and made publicly available, a useful dataset for evaluating the effectiveness of VulDeePecker and other deep learning-based vulnerability detection systems that will be developed in the future. Systematic experiments show that VulDeePecker can achieve much lower false negative rate than other vulnerability detection systems, while relieving human experts from the tedious work of manually defining features. For the 3 software products we experimented with (i.e., Xen, Seamonkey, and Libav), VulDeePecker detected 4 vulnerabilities, which were not reported in the NVD and were 'silently' patched by the vendors when they released later versions of these products.

Zhen Li and et.al [33] conducted the first comparative study to evaluate the quantitative impact of different factors on the effectiveness of deep learning-based vulnerability detection. Specifically, the main contributions of this paper are as follows. First, in order to experimentally show the effectiveness of vulnerability detection, we collect two datasets from the programs involving 126 types of vulnerabilities, while noting that the dataset published by [32] only involves two types of vulnerabilities (i.e., buffer error vulnerabilities and resource management error vulnerabilities) and only accommodates data dependency as the semantic information. One dataset contains 68,353 code gadgets (i.e., a number of statements that are semantically related to each other) with data dependency and control dependency and the other dataset contains 98,262 code gadgets with only data dependency. Second, we evaluate the quantitative impact of different factors on the effectiveness of vulnerability detection on the datasets. Some of the experimental findings are highlighted as follows: (i) Accommodating control dependency can increase the overall effectiveness of vulnerability detection F1-measure by 20.3%. (ii) The imbalanced data processing methods are not effective for the dataset we create, and the over-sampling method (e.g., SMOTE) is better than other imbalanced data processing methods for the dataset. (iii) Bidirectional Recurrent Neural Networks (RNNs) are more effective than unidirectional RNNs and convolutional neural network, which in turn are more effective than multi-layer perception. (iv) Using the last output corresponding to the time step for the BLSTM can reduce the false negative rate by 2.0% at the price of increasing the false positive rate by 0.5%. Third, we implement the deep learning-based vulnerability detection system based on an extended open source parser Joern [18] for the comparative study, while noting that VulDeePecker is implemented based on the commercial tool Checkmarx [5] which cannot accommodate new semantic information of programs. In addition, we identify important code elements in the code gadgets for vulnerability detection, which can help understand what features the deep learning model has automatically learned.

Savchenko, A. and et.al [33] presented the DeeDP system for automatic vulnerabilities detection and patch providing. DeeDP allows to detect vulnerabilities in C/C++ source code and generate patch for fixing detected issue. This system uses deep learning methods to organize rules for deciding whether a code fragment is vulnerable. Training samples were created based on source codes taken from the National Vulnerability Database (NVD), and from the NIST Software Assurance Reference Dataset (SARD). Patch generation processes can be performed based on neural network and rule-based approaches. The system uses the abstract syntax tree (AST) representations of the source code fragments. We have tested effectiveness of our approach on different open source projects.

José D'Abruzzo Pereira and et.al [34] explored the use of ML classification algorithms to detect software vulnerabilities using the output of different SATs, aiming at reducing the number of FPs without compromising the ability to detect vulnerabilities; and validate the possibility of creating ranked lists of vulnerable source code files using the output of SATs to guide the analysis by software development teams. This is particularly important in projects with a high number of files, as well as with time and budget constraints, which is the reality of most projects. To support our study we use a dataset of vulnerabilities detected by five SATs on a large set of WordPress plugins developed in PHP. We focus the study on two of the most critical vulnerability types for web application security, SQL Injection (SQLi) and Cross-Site Scripting (XSS). Comparing with the traditional 1-out-of-N (1ooN) heuristic (where an alert is raised when 1 of N detectors raises an alarm), results show an improvement for the case of SQLi vulnerabilities (namely a reduction in the false alarms). On the other hand, for XSS vulnerabilities the results using ML are equal to the ones obtained using 1ooN. Additionally, results show that it is possible to create a ranking of the source code files considering an estimation of the potential number of vulnerabilities in each file, as reported by the multiple SATs (a clear improvement when compared to the use of the alerts raised by any SAT individually).

Li, Xin and et.al [35] proposed a framework that detects software vulnerabilities in four stages: pre-processing, pre-training, representation learning, and classifier training. In the pre-processing stage, our method transforms the samples in the form of raw source code into the minimum intermediate representations through dependency analysis, program slicing, tokenization, and serialization. The length of dependencies between context is reduced by eliminating irrelevant code. The samples used in the next three stages are pre-processed respectively. In the pre-training stage, considering the lack of vulnerability samples, we conduct unsupervised learning on an extended corpus. The purpose of this process is to learn the common syntax features of program language and alleviate the OoV issue through distributed embedding. The result of the pre-training stage will serve as the parameters of the embedding layer in the next two stages. In the representation learning stage, three concatenated convolutional neural networks are utilized to obtain high-level features from a vulnerability dataset. In order to train this network, two dense layers are added in this stage. In the classifier training stage, the vulnerability dataset is transformed into high-level features using the learned network in the last stage, and a classifier is trained for vulnerability detection by the learned high-level features. Finally, test samples will be classified by the trained model. The empirical study shows that compared with the traditional methods and state-of-the-art intelligent methods, our method has made significant improvements in false positive rate, false negative rate, precision, recall, and F1 metrics.

Shashank Srikant and et.al [36] proposed a novel neural architecture - Vulnerability Classification Network (Vulcan), that we demonstrate on the problem of vulnerability classification at the line level. Vulcan takes a much more nuanced approach to forming a distributed representation than tokenization. It extracts contextual information about tokens in a line and inputs them, as Abstract Syntax Tree (AST) paths, to a bi-directional LSTM with an attention mechanism. It concurrently represents the meanings of tokens in a line by recursively embedding the lines where they are most recently defined. It has multiple 'helper' networks that transform variable length inputs to fixed lengths and sub-assembly steps performing concatenation. We experimentally evaluate Vulcan's performance and whether its distributed representation defines a latent feature space where lines of similar meaning have similar features.

Zou, Deqing and et.al [37] proposed the μ VulDeePecker, a system is in three-fold. First, a conceptual innovation underlying μ VulDeePecker is the introduction of the concept we call code attention, which can accommodate information useful for learning local features and helping pinpoint types of vulnerabilities. It refines the concept of code gadget which

is a number of statements that are semantically related to each other. Second, another innovation underlying μ VulDeePecker is redefining the concept and extraction method of code gadget by introducing control-dependence. The last innovation underlying μ VulDeePecker is a new neural network architecture. The architecture is different from the models used in previous vulnerability detection methods. It is mainly constructed from building-block BLSTM networks and aims to fuse different kinds of features from code gadget and code attention to accommodate different kinds of information. This neural network architecture may be of independent value because it provides an effective fused idea for different code features and can be referenced in other scenarios. Experimental results show that μ VulDeePecker is effective for multiclass vulnerability detection and that accommodating control-dependence (other than data-dependence) can lead to higher detection capabilities.

Hoa Khanh Dam and et.al [38] presented a novel deep learning-based approach to automatically learn features for predicting vulnerabilities in software code. We leverage LSTM to capture the long context relationships in source code where dependent code elements are scattered far apart. For example, pairs of code tokens that are required to appear together due to programming language specification (e.g. try and catch in Java) or due to API usage specification (e.g. lock() and unlock()), but that do not immediately follow each other in the textual code files. Our previous work [24] has provided a preliminary demonstration of the effectiveness of a language model based on LSTM. However, that work merely sketched the use of LSTM in predicting the next code tokens. Our current work in this paper presents a comprehensive framework where features are learned and combined in a novel way, and are then used in a novel application, i.e. building vulnerability prediction models. The learned features represent both the semantics of code tokens (semantic features) and the sequential structure of source code (syntactic features). Our automatic feature learning approach eliminates the need for manual feature engineering which occupies most of the effort in traditional approaches. Results from our experiments on 18 Java applications [8] for the Android OS platform and Firefox application [12] from public datasets demonstrate that our approach is highly effective in predicting vulnerabilities in code.

Fang Y and et.al [39] proposed as an analysis model to discover the vulnerabilities of PHP: Hypertext Preprocessor (PHP) Web programs conveniently and easily. Based on the token mechanism of PHP language, a custom tokenizer was designed, and it unifies tokens, supports some features of PHP and optimizes the parsing. Besides, the tokenizer also implements parameter iteration to achieve data flow analysis. On the Software Assurance Reference Dataset(SARD) and SQLI-LABS dataset, we trained the deep learning model of TAP by combining the word2vec model with Long Short-Term Memory (LSTM) network algorithm. According to the experiment on the dataset of CWE-89, TAP not only achieves the 0.9941 Area Under the Curve(AUC), which is better than other models, but also achieves the highest accuracy: 0.9787. Further, compared with RIPS, TAP shows much better in multiclass classification with 0.8319 Kappa and 0.0840 hamming distance.

Zhou. X and et.al [40] proposed a quantum neural network structure named QDENN for software vulnerability detection. This work is the first attempt to implement word embedding of vulnerability codes based on a quantum neural network, which proves the feasibility of a quantum neural network in the field of vulnerability detection. Experiments demonstrate that our proposed QDENN can effectively solve the inconsistent input length problem of quantum neural networks and the problem of batch processing with long sentences. Furthermore, it can give full play to the advantages of quantum computing and realize a vulnerability detection model at the cost of a small amount of measurement. Compared to other quantum neural networks, our proposed QDENN can achieve higher vulnerability detection accuracy. On the sub dataset with a small-scale interval, the model accuracy rate reaches 99%. On each subinterval data, the best average vulnerability de-

tection accuracy of the model reaches 86.3%. Compared with classical neural network, quantum neural networks can process classical information at a small memory consumption, taking advantage of the characteristics of quantum mechanics. Therefore, in addition to verifying the feasibility of vulnerability detection based on quantum neural networks, this work shows the far-reaching prospects of network security applications based on quantum neural networks. 3. In the field of large-scale vulnerability detection, vulnerability programs have longer lengths, and the lexical structure is more complex. Experimental results demonstrated that, in the field of vulnerability detection, our proposed QDENN can effectively solve the problem of inconsistent input lengths of quantum neural networks. Furthermore, it can give full play to the advantages of quantum computing and realize a vulnerability detection model at the cost of a small amount of measurement. 4. Although there have been related works on NLP based on quantum neural networks, they generate quantum circuits for each sentence based on tensor networks. However, this is not suitable for sophisticated vulnerability detection. Our proposed QDENN can also solve the problem of batch processing with long sentences.

Zimin Chen and et.al [41] proposed a deep learning-based vulnerability detector for C programs with source code, dubbed Vulnerability Deep learning-based Locator (VulDeeLocator). When compared with the state-of-the-art detector [17], VulDeeLocator offers, on average, (i) a 9.8%, 7.9%, and 8.2% improvement in the vulnerability detection F1-measure, false-positive rate, and false-negative rate respectively, and (ii) a 4.2X improvement in the vulnerability locating precision. When applied to 200 files randomly selected from three real-world software products (i.e., FFmpeg 2.8.2, Wireshark 2.0.5, and Libav 9.10), VulDeeLocator detects 18 confirmed vulnerabilities (i.e., true-positives). Among them, 16 vulnerabilities correspond to known vulnerabilities; the other two are not reported in the National Vulnerability Database (NVD) [26] but have been ‘silently’ patched by the vendor of Libav when releasing newer versions. The innovations behind VulDeeLocator are in threefold. First, we identify one root cause of the aforementioned inadequate detection capability of existing deep learningbased vulnerability detectors, and address this inadequate detection capability by linking multiple files through defineuse relations and leveraging intermediate code-based representations. The insight behind this approach is that intermediate code-based representations are in the Static Single Assignment (SSA) form and therefore can assure that each variable is defined-and-then-used and is assigned exactly once [27]. Second, we propose the notion of granularity refinement to locate the vulnerable lines of code. This principle guides us to propose a specific granularity refinement method, dubbed Bidirectional Recurrent Neural Network (BRNN) for vulnerability detection and locating’ (or BRNN-vdl for short). Although this specific method is unlikely optimal, it is effective by making VulDeeLcoator output vulnerabilities about 3 lines of code; by contrast, the input program slices to VulDeeLocator are for example 32 lines of code. Third, we prepare a vulnerability dataset in the Lower Level Virtual Machine (LLVM) intermediate code with accompanying program source code.

3.2.3. graph-based

Miltiadis Allamanis and et.al [42] proposed to use graphs to represent both the syntactic and semantic structure of code and use graph-based deep learning methods to learn to reason over program structures. It defined the VAR MISUSE task as a challenge for machine learning modeling of source code, that requires to learn (some) semantics of programs. (ii) We present deep learning models for solving the VAR NAMING and VAR MISUSE tasks by modeling the code’s graph structure and learning program representations over those graphs. (iii) We evaluate our models on a large dataset of 2.9 million lines of real-world source code, showing that our best model achieves 32.9% accuracy on the VAR NAMING task and 85.5% accuracy on the VAR MISUSE task, beating simpler baselines. (iv) We document practical relevance of VAR MISUSE by summarizing some bugs that we found in mature open-source software projects.

Hoa Khanh Dam and et.al [43] developed a new deep tree-based model for defect prediction. The model was built upon the Long Short-Term Memory (LSTM) [8], a powerful deep learning architecture to capture the long context relationships in code where dependent code elements are scattered far apart. The syntax and different levels of semantics in source code are usually represented by treebased structures such as Abstract Syntax Trees (ASTs). Hence, we adapted a tree-structured LSTM network (Tree-LSTM) [9] in which the LSTM tree in our prediction system matches exactly with the AST of an input source file, i.e. each AST node corresponds to an LSTM unit in the tree-based network. This tree-based LSTM model for source code aims to preserve both syntactic and structural information of the programs (in terms of ASTs). Through an AST node embedding mechanism, our representation of code tokens also aims to preserve their semantic relations. Our prediction system takes as input a 'raw' Abstract Syntax Tree representing a source file and predict if the file is defective or clean. The features are automatically learned through the LSTM model, thus eliminating the need for manual feature engineering which occupies most of the effort in traditional approaches. We evaluated this model using real projects provided by Samsung and the PROMISE repository.

Yaqin Zhou and et.al [44] proposed a novel graph neural network based model with composite programming representation for factual vulnerability data. This allows us to encode a full set of classical programming code semantics to capture various vulnerability characteristics. A key innovation is a new Conv module which takes as input a graph's heterogeneous node features from gated recurrent units. The Conv module hierarchically chooses more coarse features via leveraging the traditional convolutional and dense layers for graph level classification. Moreover, to both testify the potential of the composite programming embedding for source code and the proposed graph neural network model for the challenging task of vulnerability identification, we compiled manually labeled data sets from 4 popular and diversified libraries in C programming language. We name this model Devign (Deep Vulnerability Identification via Graph Neural Networks). In the composite code representation, with ASTs as the backbone, we explicitly encode the program control and data dependency at different levels into a joint graph of heterogeneous edges with each type denoting the connection regarding to the corresponding representation. The comprehensive representation, not considered in previous works, facilitates to capture as extensive types and patterns of vulnerabilities as possible, and enables to learn better node representation through graph neural networks. • We propose the gated graph neural network model with the Conv module for graph-level classification. The Conv module learns hierarchically from the node features to capture the higher level of representations for graph-level classification tasks. • We implement Devign, and evaluate its effectiveness through manually labeled data sets (cost around 600 man-hours) collected from the 4 popular C libraries. We make two datasets public together with more details (<https://sites.google.com/view/devign>). The results show that Devign achieves an average 10.51% higher accuracy and 8.68% F1 score than baseline methods. Meanwhile, the Conv module brings an average 4.66% accuracy and 6.37% F1 gain. We apply Devign to 40 latest CVEs collected from the 4 projects and get 74.11% accuracy, manifesting its usability of discovering new vulnerabilities.

Rishi Rabheru and et.al [45] In order to learn syntactic and structural properties from source code, DeepTective transforms it into a sequence of tokens to be analysed by a Gated Recurrent Unit (GRU), a neural network related to the LSTM and able to embed sequential information, in this case about the code structure. Novel to our approach for PHP, we attempt to learn semantic properties of the source code by analysing the CFG with a Graph Convolutional Network (GCN), a recent neural network architecture designed to handle graph-like data structures which during training can embed semantic and contextual information of the source code into the classification model. For our best model, this hybrid architecture achieves a 99.92% F1 score on synthetic data from SARD [16] and a 88.12% F1

score on real-world data from GitHub (our novel dataset). We investigate the impact of different dataset distributions for detecting multiple vulnerabilities, and the challenges in arXiv:2012.08835v1 [cs.CR] 16 Dec 2020 creating such datasets. The key dimensions to take into account are the nature of the samples (synthetic versus realistic), the accuracy of the labels, the balance of the classes and the overarching difficulty in generating high-quality datasets. Existing work on PHP emphasised the use of clean and synthetic datasets, and in particular SARD. We found that even a model achieving 100% F1-score when trained and tested on different portions of the same synthetic dataset can have dismal performance when tested on realistic data. We systematically compare the performance of DeepTective and a number of existing PHP vulnerability detection tools on SARD and on our real-world dataset. DeepTective outperforms the other tools on both datasets, but the gap becomes extremely large on the real-world one, even for pre-trained models. Finally, we tested DeepTective in the wild, evaluating its execution performance and its ability to generalise to a number of real-world PHP applications not present in the training dataset. We validated the practical usefulness of DeepTective by discovering 4 novel SQL injection and Cross-site scripting vulnerabilities in deployed plugins for WordPress.

Xiao Cheng and et.al [46] proposed VGDETECTOR, a new deeplearning-based code embedding approach to detecting controlflow-related (CFR) vulnerabilities. Our approach makes a new attempt by applying a recent graph convolutional network to embed code fragments in a compact and low-dimensional code representation that preserves high-level control-flow information of a vulnerable program, without the needs of manually defined rules. We have conducted our experimental evaluation based on a large-scale benchmark harvested from the Software Assurance Reference Dataset (SARD) [8], which contains a set of known real-world security flaws. To validate the effectiveness of our approach, we have compiled a list of 8,368 real-world programs (roughly 60K methods) which are only close related to CFR bugs (i.e., CWE-691, CWE-840 and CWE-438). To demonstrate the effectiveness of our approach, we have conducted extensive experiments by comparing our approach with both well-known conventional static detectors (including Flowfinder [4] and RATs [6]) and state-of-the-art machine-learning-based approaches (including Token-based embedding [33]) and Vuldeepecker [22]).

Saikat Chakraborty and et.al [47] systematically measured the generalizability of four state-of-the-art Deep Learning-based Vulnerability Prediction (hereafter DLVP) techniques [6]–[8], [12] that have been reported to detect security vulnerabilities with high accuracy (up to 95%) in the existing literature. We primarily focus on the Deep Neural Network (DNN) models that take source code as input [6]–[8], [12], [16] and detect vulnerabilities at function granularity. These models operate on a wide range of datasets that are either generated synthetically or adapted from real-world code to fit in simplified vulnerability prediction settings. First, we curate a new vulnerability dataset from two large-scale popular real-world projects (Chromium and Debian) to evaluate the performance of existing techniques in the real-world vulnerability prediction setting. The code samples are annotated as vulnerable/non-vulnerable, leveraging their issue tracking systems. Since both the code and annotations come from the real-world, detecting vulnerabilities using such a dataset reflects a realistic vulnerability prediction scenario. We also use FFMpeg Qemu dataset proposed by Zhou et al. [12]. We empirically show different shortcomings of existing datasets and models that potentially limits the usability of those techniques in practice. Our investigation found that existing datasets are too simple to represent real world vulnerabilities and existing modeling techniques do not completely address code semantics and data imbalance in vulnerability detection.

Sicong Cao and et.al [48] proposed a bidirectional graph neural network-based approach for vulnerability detection. We consider multiple code representations which can accommodate sufficient syntax and semantic information of vulnerabilities. Firstly, we

transform source code into different graphs which can reflect syntax and semantic features (e.g., Control Flow Graph). Then we vectorize these nodes in graphs as input to our Bidirectional Graph Neural Network (BGNN) for training. Finally, we extract features through a convolutional neural network and use a classifier to detect vulnerabilities. We propose BGNN4VD (Bidirectional Graph Neural Network for Vulnerability Detection), a vulnerability detection approach by constructing a Bidirectional Graph Neural-Network (BGNN). We extract the features from BGNN followed by a CNN to identify whether the program function is vulnerable. • We manually collected 2149 vulnerabilities and 3867 vulnerable functions from four popular C/C open source projects (i.e., Linux Kernel, FFmpeg, Wireshark and Libav) to construct our dataset. Compared with the state-of-the-art detectors, BGNN4VD achieves 4.9%, 11.0%, and 8.4% improvement in F1-measure, accuracy and precision, respectively.

Huanting Wang and et.al [49] introduced FUNDED, a better approach for modeling code structures. FUNDED operates on graph representations of the program source code with the capability to learn and aggregate multiple code relationships. It achieves this by leveraging the recently proposed gated graph neural networks (GGNNs) [11]. By directly operating on a graph representation, the graph neural networks (GNNs) have shown astounding successes in social networks [12], and knowledge graphs [13] and even compiled binaries [14]. While GNN provides a good starting point, applying it to develop a practical and efficient framework for software vulnerability detection is not trivial. As a standard GNN operates on a single graph representation with untyped edges, it cannot distinguish between the control and data flow information. However, such information is essential for capturing vulnerable and buggy code patterns. As demonstrated by our evaluation, when ignoring the different code relationships, a recent work that uses a vanilla GNN [15] gives marginal improvement compared to the LSTM alternatives for code vulnerability detection. FUNDED extends the GNN's capability to distinguish and model multiple code relationships (including data, control, operation order, and operand values). This is achieved by first encoding different code relationships in different relation graphs, and then using learnable, relation-specific functions to propagate and aggregate information across relation graphs. By representing the input program as multiple relation graphs with explicit control and data flows or syntactic information, our new graph model captures richer intra-program relations than prior GNN-based approaches [15]. This richer set of relationships improves the model's ability in learning useful program representation, leading to better performance of the downstream code vulnerability detection task. As we will show later, by employing the GGNN to model and distinguish the rich code relationships, our approach significantly outperforms alternative graph-based methods. We thoroughly evaluate FUNDED on large real-life datasets of code commit history and vulnerable programs written in C, Java, Php and Swift. We compare FUNDED against six state-of-the-art (SOTA) learning-based detection methods for software bugs or vulnerabilities [4, 5, 16, 6, 3, 15], and five SOTA methods for automatic vulnerable code sample collection [18, 19, 23, 20, 24]. Experimental results show that FUNDED consistently outperforms competing methods across evaluation settings, by discovering more code vulnerabilities with a lower false-positive rate.

Hantao Feng and et.al [50] First, we present a complete data processing method to convert program source code into vector representations, which can preserve all of the syntax features and reduce the data redundancy without any predefined rules or manual process. To achieve this goal, we parse the program source code into AST and slice the function node from it. Then we traverse the entire AST and map all user-defined names to fixed names. In order to learn this data by a deep neural network, we transform AST into node sequence by the preorder traversal. Using this method, we extract the function-level code, remove redundant information, and retain complete syntax information. This algorithm is very efficient in handling a large number of programs. We apply the Word2Vec[8] model to map

the node sequence into their vector representations that can be directly trained by neural networks. Second, we construct a neural network with bidirectional gate recurrent unite (Bi-GRU) to achieve efficient vulnerability detection. Our model can learn vulnerability features from the vector representations extracted from the program source code and predict vulnerabilities. By applying the pack-padded method, our model does not need any truncation and padding for the final vectors, which allows our model to handle variable-length data. Third, we collect the datasets based on Juliet Test Suite built by SARD with more than 260,000 functions to fully evaluate our model, especially the effect of the model in the face of big data. Our experiments show that our model can effectively handle big data. Compared with other vulnerability detection tools and frameworks, our model achieved high precision and low false positive rate, which is more effective than the existing methods.

Yuelong Wu and et.al [51] utilized two deep learning algorithms to automatically extract features from SCPGs. A Graph Neural Network (GNN) [10] is first used to learn graphlevel representations for SCPGs, then a Multi Layer Perceptron (MLP) is employed to extract features automatically. The contributions of this paper are as follows: , The SCPG is proposed to represent the source code at the software function level, which enables fine-grained vulnerability detection. The SCPG representation approach has the advantages of conserving both syntactic and semantic information of source code while keeping itself small enough for computing. , An effective and automatic feature extraction approach is proposed. It utilizes GNN and MLP to automatically learn graph representation and extract features. In comparison to Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) approaches, the proposed approach is suitable to learn representations for SCPGs because of the intrinsic ability of GNNs to learn from graphs. The comparison experimental results indicate that the proposed approach achieves scores of 83.6%, 83.1% on F1-score, recall respectively, which are higher than existing CNN, RNN approaches; it also achieves the best accuracy of 89.2% and an acceptable precision of 84.0%.

Zou, Deqing and Hu et.al [52] aimed to combine the fine-granularity of the slice-level with the complete semantics of the functionlevel to construct an accurate and interpretable vulnerability detector. Specifically, we mainly address two challenges. • How to combine the advantages of slice-level and functionlevel vulnerability detection to reduce false positives and false negatives? • How to give a detailed interpretation of the vulnerability detection results? To address the first challenge, we leverage two deep neural networks to perform vulnerability detection at both function-level and slice-level. Given the source code of a function, we first apply static analysis to extract the Program Dependency Graph (PDG). Then we split PDG into subgraphs (i.e., slices) via program slicing based on the interesting points of vulnerabilities. These slices will be fed into a trained slice-level detection model to calculate their predicted probabilities. The lower the probability, the less likely it is to contain vulnerable code. Therefore, we discard those slices with the lowest probability. Based on this, we can purify the function semantics to achieve more accurate vulnerability detection at function level. To address the second challenge, we incorporate an attention mechanism into the function-level vulnerability detection model to determine the weight of each slice. Meanwhile, we also consider the probability of the slice predicted by the slice-level vulnerability detector. After combining the weight with the probability, we can compute a specific score for each slice. In this way, each statement is assigned a score by accumulating the slice scores to which it belongs. The higher the score of the statement, the more likely it is a vulnerable statement. The experimental results indicate that mVulPreter outperforms existing state-of-the-art vulnerability detection approaches (i.e., Checkmarx, FlawFinder, RATS, TokenCNN, StatementLSTM, SySeVR, and Devign).

Van-Anh Nguyen and et.al [53] developed a general, practical, and programming language-independent model capable of running on various source codes and libraries without difficulty. Therefore, we consider vulnerability detection as an inductive text classi-

fication problem and propose ReGVD, a simple yet effective graph neural network-based model for the problem. In particular, ReGVD views each raw source code as a flat sequence of tokens to build a graph, wherein node features are initialized by only the token embedding layer of a pre-trained programming language (PL) model. ReGVD then leverages residual connection among GNN layers and examines a mixture of graph-level sum and max poolings to return a graph embedding for the source code. To demonstrate the effectiveness of ReGVD, we conduct extensive experiments to compare ReGVD with the strong and up-to-date baselines on the benchmark vulnerability detection dataset from CodeXGLUE. Experimental results show that the proposed ReGVD is significantly better than the baseline models and obtains the highest accuracy of 63.69% on the benchmark dataset. ReGVD can be seen as a general, practical, and programming language-independent model that can run on various source codes and libraries without difficulty.

3.2.4. image-based

Patrick Keller and et.al [54] proposed to investigate another representation learning direction for capturing semantics. In contrast to recent works which all focus on lexical and syntactical information to capture semantics, the intuition behind our approach is to mimic the way a human would instantly perceive code. The data is received through visual perception and forms an image in the head of the programmer. This image is then analyzed for structures the programmer has seen before, by applying his experience. From those recognized structures, the programmer may identify patterns of functionality implementations, which would help him to rapidly infer the semantics of the code fragment, leading to a general understanding of the code. We apply the same methodology to design the WySiWiM ('What You See Is What It Means') approach: instead of directly training a complex and opaque semantic representation or embeddings based on syntactical information in source code, we simply render source code into a visual representation that is natural to code. This step is supposed to model the human visual perception. After the visualization process, WySiWiM performs two different procedures. First, the visual structures of the code are extracted. To that end, a pre-trained image classification neural network, i.e., a neural network that has been trained on other image classification datasets 1 is used to yield a vector of internal features which represent structural information of the input image (i.e., of the code). Optionally, the pre-trained network can be re-trained by adding samples of images representing code. Second, the feature vectors are used for learning to discriminate between samples implementing different semantics, just as a human developer would do. with experiments on the BigCloneBench (Java) and Open Judge (C) datasets that although simple, our WySiWiM approach performs as effectively as state of the art approaches such as ASTNN or TBCNN.

Yueming Wu and et.al [55] How to efficiently convert the source code of a function into an image while preserving the program details? To tackle the challenge, we first conduct program analysis to distill the program semantics of a function into a program dependencygraph (PDG) which contains both control-flowand dataflow details of source code. After obtaining the PDG of a function, we treat it as a social network and apply centrality analysis on the network to attach the graph structural information to each line of code. In social network analysis, centrality analysis [25 , 31] has been proposed to measure the importance of a node within the network. Specifically, we leverage three different centralities (i.e., degree centrality [25], katz centrality [31], and closeness centrality [25]) to commence our image transformation. There are two main reasons for using three centralities. First, different centralities can maintain the graph properties from different aspects [52]. Second, an image generally has three channels (i.e., red, green, and blue) and they work together to produce a complete image. The output of centrality analysis is an image while preserving the graph details from three aspects. Given generated images, we then train a Convolutional Neural Network (CNN) [34 , 36] model and use it to detect vulnerability. To pinpoint the vulnerable lines of code in a function, we use a deep

visualization technique (i.e., Class Activation Map [20]) on our images to obtain the corresponding heatmaps, these heatmaps can help security analysts understand why the function is labeled as a vulnerability. We implement VulCNN and evaluate it on a dataset of 40,657 functions which consists of 13,687 vulnerable functions and 26,970 non-vulnerable functions. Evaluation results show that VulCNN can achieve better effectiveness than eight comparative vulnerability detectors (i.e., Checkmarx [5], FlawFinder [6], RATS [12], TokenCNN [47], VulDeePecker [41], SySeVR [40], VulDeeLocator [38], and Devign [56]). Furthermore, VulCNN is more than six times faster than another state-of-the-art graph-based vulnerability detection tool (i.e., Devign). To validate the ability of VulCNN on large-scale vulnerability scanning, we conduct a case study on more than 25 million lines of code. Through the scanning reports, we finally discover 73 vulnerabilities that are not reported in NVD.

4. Software Source code vulnerability static repair based on machine learning

4.1. bug-based

4.2. vulnerability-based

4.3. seq-based

Rahul Gupta and et.al [57] presented an end-to-end solution, called DeepFix, that does not use any external tool to localize or fix errors. We use a compiler only to validate the fixes suggested by DeepFix. At the heart of DeepFix is a multi-layered sequenceto-sequence neural network with attention (Bahdanau, Cho, and Bengio 2014), comprising of an encoder recurrent neural network (RNN) to process the input and a decoder RNN with attention that generates the output. The network is trained to predict an erroneous program location along with the correct statement. DeepFix invokes it iteratively to fix multiple errors in the program one-by-one. We apply DeepFix on C programs written by students for 93 different programming tasks in an introductory programming course. The nature of these tasks and thereby, that of the programs vary widely. Nevertheless, our network generalizes well to programs from across these tasks. Out of 6971 erroneous programs, DeepFix fixed 1881 (27%) programs completely and 1338 (19%) programs partially.

Michele Tufano and et.al [56] started by mining a large set of (787k) bug-fixing commits from GitHub. From these commits, we extract method-level AST edit operations using fine-grained source code differencing. We identify multiple method-level differences per bug-fixing commit and independently consider each one, yielding to 2 . 3 M bug-fix pairs (BFPs). After that, the code of the BFPs is abstracted to make it more suitable for the NMT model. Finally, an encoder-decoder model is used to understand how the buggy code is transformed into fixed code. Once the model has been trained, it is used to generate patches for unseen code. We empirically investigate the potential of NMT to generate candidate patches that are identical to the ones implemented by developers. The results indicate that trained NMT model is able to successfully predict the fixed code, given the buggy code, in 9% of the cases.

Eddie Antonio Santos and et.al [57] proposed a methodology that locates where syntax errors occur, and suggests possible changes to the token stream that can fix the error identified. This methodology finds syntax errors by using language models trained on correct source code to find tokens that seem out of place. Fixes are synthesized by consulting the language models to determine what tokens are more likely at the estimated error location. We compare n-gram and LSTM (long short-term memory) language models for this task, each trained on a large corpus of Java code collected from GitHub. Unlike prior work, our methodology does not rely that the problem source code comes from the same domain as the training data. We evaluated against a repository of real student mistakes. Our tools are able to find a syntactically-valid fix within its top2 suggestions, often producing the exact fix that the student used to resolve the error. The results show that this tool and methodology can locate and suggest corrections for syntax errors. Our methodology is of practical use to all programmers, but will be especially useful to novices frustrated with

incomprehensible syntax errors.

Jacob Harer and et.al [58] addressed this problem is adversarial learning with Generative Adversarial Networks (GANs) [9]. This approach allows us to train without paired examples. We employ a traditional NMT model as the generator, and replace the typical negative likelihood loss with the gradient stemming from the loss of an adversarial discriminator. The discriminator is trained to distinguish between NMT-generated outputs and real examples of desired output, and so its loss serves as a proxy for the discrepancy between the generated and real distributions. This problem has three main difficulties. Firstly, sampling from the output of NMT systems, in order to produce discrete outputs, is non-differentiable. We address this problem by using a discriminator which operates directly on the expected (soft) outputs of the NMT system during training, which we thoroughly discuss in Section 3.2. Secondly, adversarial training does not guarantee that the generated code will correspond to the input bad code (i.e. the generator is trained to match distributions, not samples). To enforce the generator to generate useful repairs, (i.e., generated code is a repaired version of input bad code), we condition our NMT generator on the input x by incorporating two novel generator loss functions, described in Section 3.3. Thirdly, the domains we consider are not bijective, i.e., a bad code can have more than one repair or a good code can be broken in more than one way. The regularizers we propose in Section 3.3 still work in this case. We should note that although our motivation is to repair source code, the approach and the techniques proposed in this paper are application-agnostic in that they can be applied to other similar problems, such as correcting grammar errors or converting between negative and positive sentiments (e.g., in online reviews.). Additionally, while software vulnerability repair is a harder problem than detection, our proposed repair technique can leverage the same datasets used for detection and yields a much more explainable and useful tool than detection alone.

Venkatesh Theru Mohan and et.al [59] proposed neural network model is neither trained with any defect datasets nor bug-free programming source codes, instead it is trained using structural semantic details of Abstract Syntax Tree (AST) where each node represents a construct appearing in the source code. The type of the undeclared variables is also inferred by performing type binding using the semantic elements of AST representation that provides about the type information of the variables thereby saving the compiler's time in performing the type binding of those undeclared variables. The comprehensive information of the ASTs, the motive of Long Short-Term Memory (LSTM), detailed view of the training approach and implementation, the generation approach and different scenarios where undeclared variables will be caused and possible cases of type inference of them are be discussed. This model is implemented to fix of the most common semantic errors, such as undeclared variable errors as well as infer their type information before program compilation. By this approach, the model has achieved in correctly locating and identifying 81% of the programs on prutor dataset of 1059 programs with only undeclared variable errors and also inferring their types correctly in 80% of the programs.

Hideaki Hata and et.al[60] proposed Ratchet, a NMT-based technique that generates bug fixes based on prior bug-and-fix examples. To evaluate the effectiveness of the technique, we perform an empirical study with five large software projects, namely Ambari, Camel, Hadoop, Jetty and Wicket. We use the pattern-based patch suggestion inspired by the Plastic Surgery work [51], as a comparison baseline and examine the effectiveness of our NMT-based technique. In particular, we quantify the number of cases where our NMTbased technique is able to generate a valid fix and how accurate the generated fixes are. Our findings showed that Ratchet is able to generate a valid statements in 98.7% of the cases and achieves an F1 measure between 0.29 – 0.83 with respect to the actual fixes adopted in the code base. For all five projects, Ratchet was able to either outperform or

perform as well as the baseline.

Hossein Hajipour and et.al [61] proposed a deep generative framework to automatically correct programming errors by learning the distribution of potential fixes. At the core of our approach is a conditional variational autoencoder that is trained to sample accurate and diverse fixes for the given erroneous programs, and interacts with a compiler that evaluates the sampled candidate fixes in the context of the given programs. The interplay of sampler and compiler allows for an iterative refinement procedure. A key contribution of our approach is a novel regularizer which encourages diversity by penalizing the distance among candidate samples, which thereby significantly increases the effectiveness by producing more diverse samples. Furthermore, experiments show that the steady increase in discovery of correct programs as we draw more candidate fixes is a strong indication that our proposed model can be the basis of further advances in debugging by sampling based approaches.

Marko Vasic and et.al [62] presented a model that jointly learns to perform: 1) classification of the program as either faulty or correct (with respect to VARMISUSE bugs), 2) localization of the bug when the program is classified as faulty, and 3) repair of the localized bug. One of the key insights of our joint model is the observation that, in a program containing a single VARMISUSE bug, a variable token can only be one of the following: 1) a buggy variable (the faulty location), 2) some occurrence of the correct variable that should be copied over the incorrect variable into the faulty location (a repair location), or 3) neither the faulty location nor a repair location. This arises from the fact that the variable in the fault location cannot contribute to the repair of any other variable – there is only one fault location – and a variable in a repair location cannot be buggy at the same time. This observation leads us to a pointer model that can point at locations in the input (Vinyals et al., 2015) by learning distributions over input tokens. The hypothesis that a program that contains a bug at a location likely contains ingredients of the repair elsewhere in the program (Engler et al., 2001) has been used quite effectively in practice (Le Goues et al., 2012). Mechanisms based on pointer networks can play a useful role to exploit this observation for repairing programs. We compare our joint prediction model to an enumerative approach for repair. Our results show that the joint model not only achieves a higher classification, localization, and repair accuracy, but also results in high true positive score.

Shan Huang and et.al [citer-14-fun-s-GRU-n-(C-Java)-SARD] showed that seq2seq models, successful in natural language correction, are also applicable in programming language correction. Our results show seq2seq models can be well applied in providing suggestions to potential errors and have a decent correct rate (above 70% in C/C dataset and above 50% in Java dataset) in code auto-correction. Although these results are only limited in Juliet Test Suites, we expect that, given sufficient training data, seq2seq models can also perform well when applied on other PLC problems. Based on the commonly used encoder-decoder structure, we introduce a general pyramid encoder in seq2seq models. Our results demonstrate that this structure significantly reduces the memory cost and computational cost. This is helpful because Programming language correction (PLC) are generally more computationally expensive than natural language correction (NLC), due to its longer average instance length.

Daya Guo and et.al [63] presented GraphCodeBERT, a pre-trained model for programming language that considers the inherent structure of code. Instead of taking syntactic-level structure of code like abstract syntax tree (AST), we leverage semantic-level information of code, i.e. data flow, for pretraining. Data flow is a graph, in which nodes represent variables and edges represent the relation of ‘where-the-value-comes-from’ between variables. Compared with AST, data flow is less complex and does not bring an unnecessarily deep hierarchy, the property of which makes the model more efficient. In order to learn code representation from source code and code structure, we introduce two new structure-

aware pre-training tasks. One is data flow edges prediction for learning representation from code structure, and the other is variable-alignment across source code and data flow for aligning representation between source code and code structure. GraphCodeBERT is based on Transformer neural architecture and we extend it by introducing a graph-guided masked attention function to incorporate the code structure. Experimental results show that the Transformer significantly outperforms LSTM. Results in the second group shows that pre-trained models outperform Transformer models further, and GraphCodeBERT achieves better performance than other pre-trained models on both datasets, which shows leveraging code structure information are helpful to the task of code refinement.

Zimin Chen and et.al [64] proposed end-to-end program repair approach is called S EQUENCE R and it works as follows. First, we focus on oneline fixes: we predict the fixed version of a buggy programming line. For this, we create a carefully curated training and testing dataset of one-line commits. Second, we devise a sequence-to-sequence network architecture that is specifically designed to address the two main aforementioned challenges. To address the unlimited vocabulary problem, we use the copy mechanism [9]; this allows S EQUENCE R to predict the fixed line, even if the fix contains a token that was too rare (i.e., an API call that appears only in few cases, or a rare identifier used only in one class) to be considered in the vocabulary. This copy mechanism works even if the fixed line should contain tokens which were not in the training set. To address the dependency problem, we construct abstract buggy context from the buggy class, which captures the most important context around the buggy source code and reduces the complexity of the input sequence. This enables us to capture long range dependencies that are required for the fix. SEQUENCER is able to perfectly predict the fixed line for 950/4,711 testing samples, and find correct patches for 14 bugs in Defects4J benchmark.

4.4. graph-based

Siqi Ma and et.al [65] designed and implemented a novel tool, called VuRLE (Vulnerability Repair by Learning from Examples), that can help developers automatically detect and repair multiple types of vulnerabilities. VuRLE works as follows: 1. VuRLE analyzes a training set of repair examples and identifies edit blocks – each being series of related edits and its context from each example. Each example contains a vulnerable code and its repaired code. 2. VuRLE clusters similar edit blocks in to groups. 3. Next, VuRLE generates several repair templates for each group from pairs of highly similar edits. 4. VuRLE then uses the repair templates to identify vulnerable code. 5. VuRLE eventually selects a suitable repair template and applies the transformative edits in the template to repair a vulnerable code. VuRLE addresses the first limitation of LASE by generating many repair templates instead of only one. These templates are put into groups and are used collectively to accurately identify vulnerabilities. VuRLE also employs a heuristics that identifies the most appropriate template for a detected vulnerability. It addresses the second limitation by breaking a repair example into several code segments. It then extracts an edit block from each of the code segment. These edit blocks may cover different bugs and can be used to generate different repair templates. This will result in many edit blocks though, and many of which may not be useful in the identification and fixing of vulnerabilities. To deal with this issue, VuRLE employs a heuristics to identify suitable edit blocks that can be generalized into repair templates. We evaluate VuRLE on 279 vulnerabilities from 48 real-world applications using 10-fold cross validation setting. In this experiment, VuRLE successfully detects 183 (65.59%) out of 279 vulnerabilities, and repairs 101 of them. This is a major improvement when compared to LASE, as it can only detects 58 (20.79%) out of the 279 vulnerabilities, and repairs 21 of them.

Saikat Chakraborty and et.al[66] designed a two step encoder-decoder model that models the probability distribution of changes. In the first step, it learns to suggest structural changes in code using a tree-to-tree model, suggesting structural changes in the form

of Abstract Syntax Tree (AST) modifications. Tree-based models, unlike their token-based counterparts, can capture the rich structure of code and always produce syntactically correct patches. In the second step, the model concretizes the previously generated code fragment by predicting the tokens conditioned on the AST that was generated in the first step: given the type of each leaf node in the syntax tree, our model suggests concrete tokens of the correct type while respecting scope information. We combine these two models to realize C ODIT , a code change suggestion engine, which accepts a code fragment and generates potential edits of that snippet.

Liwei Wu and et.al [67] challenge is that the traditional Recurrent Neural Network(RNN)[16, 19] encoder can not effectively encode the code sequence . they proposed a novel deep supervised learning model, called Graph-based Grammar Fix(GGF). GGF treats the code as a graph. When parsing the erroneous code, the parser may crash but some parts of the AST have been created which is called sub-AST in this paper. The graph is built with sub-AST structure information. Identical variable identifiers are also connected in the graph. A graph example is shown in Fig. 3. Since the parser may crash in the parsing process, there may be some isolated points and some error edges in this graph. To tackle the problem, GGF treats the code as a mixture of graphs and token sequences. Specifically, GGF uses both Gated Recurrent Unit (GRU, a variant of RNN)[9] and Gated Graph Neural Networks (GGNN, a kind of graph encoder)[21] as encoders. challenge is how to fix the code. propose a new token replacement mechanism which can locate the error and predict the correct token. The error is located by an open interval which has two cases. If the interval contains one token, GGF will replace or delete the token here. If the interval is empty, GGF will insert a token here. Compared to the seq-to-seq architecture, GGF only changes one token at a time and does not need to predict the whole correct texts. When more than one error candidate is found, GGF will pick the candidate with the highest probability and fix it. This could lead to the elimination of other error candidates in the next iteration. However, in the seq-to-seq architecture, the model needs to fix all error candidates at once in the decoding process and this may introduce additional errors in previous decoding steps. Compared to the line replacement mechanism, our method does not need to leverage the text structure which makes our model independent of the coding style in the training dataset. Compared to the token replacement mechanism proposed by [27], our method predicts the probability of different actions so does not need to try the three action types and can fix code containing multiple errors more efficiently.

Michihiro Yasunaga and et.al [68] present DrRepair, a novel approach to program repair that addresses these two challenges. Our key innovations are two-fold: 1) modeling of program repair with program-feedback graphs and 2) self-supervised learning with unlabeled programs. Program repair requires reasoning jointly over the symbols (e.g. identifiers, types, operators) across source code and diagnostic feedback. It's a self-supervised learning paradigm for program repair that leverages unlabeled programs to create a large amount of extra training data. Specifically, we collect working programs from online resources related to our problem domain (programming contests in our case), and design a procedure that corrupts a working program into a broken one, thereby generating new examples of h broken program, fixed program i . In our experiments, we prepare extra data about 10 times the size of original datasets in this way, use it to pre-train our models, and fine-tune on the target task. Experimental results show that our approach (DrRepair) outperforms prior work significantly, achieving 68.2% full repair on the DeepFix test set (22.9% absolute over the prior best), and 48.4% synthesis success rate on the SPoC test set (+3.7% absolute over the prior best at the time of this work).

Elizabeth Dinella and et.al citer-13-fun-g-(one-hot)-n-Javascript-github targets bugs that are more diverse and complex in nature (i.e. bugs that require adding or deleting statements to fix). proposed a novel learning-based approach for finding and fixing bugs

in Javascript programs automatically. Javascript is a scripting language designed for web application development. It has been the most popular programming language on GitHub since 2014 (Github, 2019). Repairing Javascript code presents a unique challenge as bugs manifest in diverse forms due to unusual language features and the lack of tooling support. Therefore, the primary goal of our approach is generality since it must be effective against a board spectrum of programming errors, such as using wrong operators or identifiers, accessing undefined properties, mishandling variable scopes, triggering type incompatibilities, among many others. Another important novel aspect concerns our approach's ability to deal with bugs that are more complex and semantic in nature, namely, bugs that require adding or removing statements from a program, which are not considered by prior works. Finally, compared to automated program repair techniques which require knowledge of bug location, this paper presents an end-to-end approach including localizing bugs, predicting the types of fixes, and generating patches. By training on 290,715 Javascript code change commits collected from Github, H OPPITY correctly detects and fixes bugs in 9,490 out of 36,361 programs using a beam size of three.

5. Results

This section may be divided by subheadings. It should provide a concise and precise description of the experimental results, their interpretation as well as the experimental conclusions that can be drawn.

5.1. Subsection

5.1.1. section

Bulleted lists look like this:

- First bullet;
- Second bullet;
- Third bullet.

Numbered lists can be added as follows:

1. First item;
2. Second item;
3. Third item.

The text continues here.

5.2. Figures, Tables and Schemes

All figures and tables should be cited in the main text as Figure 1, Table 2, Table 3, etc.



Figure 1. This is a figure. Schemes follow the same formatting. If there are multiple panels, they should be listed as: **(a)** Description of what is contained in the first panel. **(b)** Description of what is contained in the second panel. Figures should be placed in the main text near to the first time they are cited. A caption on a single line should be centered.

Table 2. This is a table caption. Tables should be placed in the main text near to the first time they are cited.

Title 1	Title 2	Title 3
Entry 1	Data	Data
Entry 2	Data	Data

Table 3. This is a wide table.

Title 1	Title 2	Title 3	Title 4
Entry 1	Data	Data	Data
Entry 2	Data	Data	Data ¹

¹ This is a table footnote.

Text.

Text.

1269

1270

5.3. *Formatting of Mathematical Components*

1271

This is the example 1 of equation:

1272

$$a = 1,$$

(1)

the text following an equation need not be a new paragraph. Please punctuate equations as regular text.

1273

1274

This is the example 2 of equation:

1275

$$a = b + c + d + e + f + g + h + i + j + k + l + m + n + o + p + q + r + s + t + u + v + w + x + y + z$$

(2)



Figure 2. This is a wide figure.

Please punctuate equations as regular text. Theorem-type environments (including propositions, lemmas, corollaries etc.) can be formatted as follows:

Theorem 1. *Example text of a theorem.*

The text continues here. Proofs must be formatted as follows:

Proof of Theorem 1. Text of the proof. Note that the phrase “of Theorem 1” is optional if it is clear which theorem is being referred to. □

The text continues here.

6. Conclusions

This section is not mandatory, but can be added to the manuscript if the discussion is unusually long or complex.

7. Challenges and Future trends

Authors should discuss the results and how they can be interpreted from the perspective of previous studies and of the working hypotheses. The findings and their implications should be discussed in the broadest context possible. Future research directions may also be highlighted.

Author Contributions: For research articles with several authors, a short paragraph specifying their individual contributions must be provided. The following statements should be used “Conceptualization, X.X. and Y.Y.; methodology, X.X.; software, X.X.; validation, X.X., Y.Y. and Z.Z.; formal analysis, X.X.; investigation, X.X.; resources, X.X.; data curation, X.X.; writing—original draft preparation, X.X.; writing—review and editing, X.X.; visualization, X.X.; supervision, X.X.; project administration, X.X.; funding acquisition, Y.Y. All authors have read and agreed to the published version of the manuscript.”, please turn to the [CRediT taxonomy](#) for the term explanation. Authorship must be limited to those who have contributed substantially to the work reported.

Funding: Please add: “This research received no external funding” or “This research was funded by NAME OF FUNDER grant number XXX.” and “The APC was funded by XXX”. Check

carefully that the details given are accurate and use the standard spelling of funding agency names at <https://search.crossref.org/funding>, any errors may affect your future funding.

Institutional Review Board Statement: In this section, you should add the Institutional Review Board Statement and approval number, if relevant to your study. You might choose to exclude this statement if the study did not require ethical approval. Please note that the Editorial Office might ask you for further information. Please add “The study was conducted in accordance with the Declaration of Helsinki, and approved by the Institutional Review Board (or Ethics Committee) of NAME OF INSTITUTE (protocol code XXX and date of approval).” for studies involving humans. OR “The animal study protocol was approved by the Institutional Review Board (or Ethics Committee) of NAME OF INSTITUTE (protocol code XXX and date of approval).” for studies involving animals. OR “Ethical review and approval were waived for this study due to REASON (please provide a detailed justification).” OR “Not applicable” for studies not involving humans or animals.

Informed Consent Statement: Any research article describing a study involving humans should contain this statement. Please add “Informed consent was obtained from all subjects involved in the study.” OR “Patient consent was waived due to REASON (please provide a detailed justification).” OR “Not applicable” for studies not involving humans. You might also choose to exclude this statement if the study did not involve humans.

Written informed consent for publication must be obtained from participating patients who can be identified (including by the patients themselves). Please state “Written informed consent has been obtained from the patient(s) to publish this paper” if applicable.

Data Availability Statement: In this section, please provide details regarding where data supporting reported results can be found, including links to publicly archived datasets analyzed or generated during the study. Please refer to suggested Data Availability Statements in section “MDPI Research Data Policies” at <https://www.mdpi.com/ethics>. If the study did not report any data, you might add “Not applicable” here.

Acknowledgments: In this section you can acknowledge any support given which is not covered by the author contribution or funding sections. This may include administrative and technical support, or donations in kind (e.g., materials used for experiments).

Conflicts of Interest: Declare conflicts of interest or state “The authors declare no conflict of interest.” Authors must identify and declare any personal circumstances or interest that may be perceived as inappropriately influencing the representation or interpretation of reported research results. Any role of the funders in the design of the study; in the collection, analyses or interpretation of data; in the writing of the manuscript; or in the decision to publish the results must be declared in this section. If there is no role, please state “The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results”.

Sample Availability: Samples of the compounds ... are available from the authors.

Abbreviations

The following abbreviations are used in this manuscript:

MDPI Multidisciplinary Digital Publishing Institute

DOAJ Directory of open access journals

TLA Three letter acronym

LD Linear dichroism

Appendix A

Appendix A.1

The appendix is an optional section that can contain details and data supplemental to the main text—for example, explanations of experimental details that would disrupt the flow of the main text but nonetheless remain crucial to understanding and reproducing the research shown; figures of replicates for experiments of which representative data are shown in the main text can be added here if brief, or as Supplementary Data. Mathematical proofs of results not central to the paper can be added as an appendix.

Table A1. This is a table caption.

Title 1	Title 2	Title 3
Entry 1	Data	Data
Entry 2	Data	Data

Appendix B

All appendix sections must be cited in the main text. In the appendices, Figures, Tables, etc. should be labeled, starting with “A”—e.g., Figure A1, Figure A2, etc.

- Common Vulnerabilities and Exposures. In Proceedings of the <https://cve.mitre.org>.
- National Vulnerability Database. In Proceedings of the <https://nvd.nist.gov/>.
- 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*, 56:1–56:36. <https://doi.org/10.1145/3092566>.
- Jiang, J.; Yu, X.; Sun, Y.; Zeng, H. A Survey of the Software Vulnerability Discovery Using Machine Learning Techniques. In Proceedings of the Artificial Intelligence and Security - 5th International Conference, ICAIS 2019, New York, NY, USA, July 26–28, 2019, Proceedings, Part IV; Sun, X.; Pan, Z.; Bertino, E., Eds. Springer, 2019, Vol. 11635, *Lecture Notes in Computer Science*, pp. 308–317. https://doi.org/10.1007/978-3-030-24268-8_29.
- Semasaba, A.O.A.; Zheng, W.; Wu, X.; Agyemang, S.A. Literature survey of deep learning-based vulnerability analysis on source code. *IET Softw.* **2020**, *14*, 654–664. <https://doi.org/10.1049/iet-sen.2020.0084>.
- Lin, G.; Wen, S.; Han, Q.; Zhang, J.; Xiang, Y. Software Vulnerability Detection Using Deep Neural Networks: A Survey. *Proc. IEEE* **2020**, *108*, 1825–1848. <https://doi.org/10.1109/JPROC.2020.2993293>.
- Wang, J.; Huang, M.; Nie, Y.; Li, J. Static Analysis of Source Code Vulnerability Using Machine Learning Techniques: A Survey. In Proceedings of the 2021 4th International Conference on Artificial Intelligence and Big Data (ICAIBD). IEEE, 2021, pp. 76–86.
- Hanif, H.; Nasir, M.H.N.B.M.; Razak, M.F.A.; Firdaus, A.; Anuar, N.B. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *J. Netw. Comput. Appl.* **2021**, *179*, 103009. <https://doi.org/10.1016/j.jnca.2021.103009>.
- Senanayake, J.; Kalutarage, H.; Al-Kadri, M.O.; Petrovski, A.; Piras, L. Android source code vulnerability detection: a systematic literature review. *ACM Computing Surveys (CSUR)* **2022**.
- Pooja, S.; Chandrakala, C.; Raju, L.K. Developer’s Roadmap to design Software Vulnerability Detection Model using different AI approaches. *IEEE Access* **2022**, *10*, 75637–75656.
- Marjanov, T.; Pashchenko, I.; Massacci, F. Machine Learning for Source Code Vulnerability Detection: What Works and What Isn’t There Yet. *IEEE Security & Privacy* **2022**, *20*, 60–76.
- Goues, C.L.; Pradel, M.; Roychoudhury, A. Automated program repair. *Commun. ACM* **2019**, *62*, 56–65. <https://doi.org/10.1145/3318162>.
- Gazzola, L.; Micucci, D.; Mariani, L. Automatic Software Repair: A Survey. *IEEE Trans. Software Eng.* **2019**, *45*, 34–67. <https://doi.org/10.1109/TSE.2017.2755013>.
- Shen, Z.; Chen, S. A Survey of Automatic Software Vulnerability Detection, Program Repair, and Defect Prediction Techniques. *Secur. Commun. Networks* **2020**, *2020*, 8858010:1–8858010:16. <https://doi.org/10.1155/2020/8858010>.
- Li, Z.; Zou, D.; et.al. VulPecker: an automated vulnerability detection system based on code similarity analysis. In Proceedings of the Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5–9, 2016; Schwab, S.; Robertson, W.K.; Balzarotti, D., Eds. ACM, 2016, pp. 201–213.
- Akram, J.; Qi, L.; Luo, P. VCIPR: Vulnerable Code is Identifiable When a Patch is Released (Hacker’s Perspective). In Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi’an, China, April 22–27, 2019. IEEE, 2019, pp. 402–413. <https://doi.org/10.1109/ICST.2019.00049>.
- Akram, J.; Luo, P. SQVDT: A scalable quantitative vulnerability detection technique for source code security assessment. *Softw. Pract. Exp.* **2021**, *51*, 294–318. <https://doi.org/10.1002/spe.2905>.
- Sun, H.; Cui, L.; Li, L.; Ding, Z.; Hao, Z.; Cui, J.; Liu, P. VDSimilar: Vulnerability detection based on code similarity of vulnerabilities and patches. *Comput. Secur.* **2021**, *110*, 102417. <https://doi.org/10.1016/j.cose.2021.102417>.
- Salimi, S.; Kharrazi, M. VulSlicer: Vulnerability detection through code slicing. *J. Syst. Softw.* **2022**, *193*, 111450. <https://doi.org/10.1016/j.jss.2022.111450>.
- Yamaguchi, F.; Lottmann, M.; Rieck, K. Generalized vulnerability extrapolation using abstract syntax trees. In Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3–7 December 2012; Zakon, R.H., Ed. ACM, 2012, pp. 359–368. <https://doi.org/10.1145/2420950.2421003>.
- Zhang, X.; Sun, H.; He, Z.; Gu, M.; Feng, J.; Zhang, Y. VDBWGD: Vulnerability Detection Based On Weight Graph And Deep Learning. In Proceedings of the 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN

- Workshops 2022, Baltimore, MD, USA, June 27–30, 2022. IEEE, 2022, pp. 186–190. <https://doi.org/10.1109/DSN-W54100.2022.00039>. 1399 1400
22. Yang, L.; Li, X.; Yu, Y. VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes. In Proceedings of the 2017 IEEE Global Communications Conference, GLOBECOM 2017, Singapore, December 4–8, 2017. IEEE, 2017, pp. 1–7. <https://doi.org/10.1109/GLOCOM.2017.8254428>. 1401 1402 1403
 23. Gawron, M.; Cheng, F.; Meinel, C. Automatic Vulnerability Classification Using Machine Learning. In Proceedings of the Risks and Security of Internet and Systems - 12th International Conference, CRISIS 2017, Dinard, France, September 19–21, 2017, Revised Selected Papers; Cuppens, N.; Cuppens, F.; Lanet, J.; Legay, A.; García-Alfaro, J., Eds. Springer, 2017, Vol. 10694, *Lecture Notes in Computer Science*, pp. 3–17. https://doi.org/10.1007/978-3-319-76687-4_1. 1404 1405 1406 1407
 24. Chernis, B.; Verma, R.M. Machine Learning Methods for Software Vulnerability Detection. In Proceedings of the Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics, IWSPA@CODASPY 2018, Tempe, AZ, USA, March 19–21, 2018; Verma, R.M.; Kantarcioglu, M., Eds. ACM, 2018, pp. 31–39. <https://doi.org/10.1145/3180445.3180453>. 1408 1409 1410
 25. Du, X.; Chen, B.; et.al. Leopard: identifying vulnerable code for vulnerability assessment through program metrics. In Proceedings of the Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019; Atlee, J.M.; Bultan, T.; Whittle, J., Eds. IEEE / ACM, 2019, pp. 60–71. <https://doi.org/10.1109/ICSE.2019.00024>. 1411 1412 1413
 26. Sahin, C.B.; Abualigah, L.M. A novel deep learning-based feature selection model for improving the static analysis of vulnerability detection. *Neural Comput. Appl.* **2021**, *33*, 14049–14067. <https://doi.org/10.1007/s00521-021-06047-x>. 1414 1415
 27. Shuai, B.; Li, H.; Li, M.; Zhang, Q.; Tang, C. Automatic classification for vulnerability based on machine learning. In Proceedings of the IEEE International Conference on Information and Automation, ICIA 2013, Yinchuan, China, August 26–28, 2013. IEEE, 2013, pp. 312–318. <https://doi.org/10.1109/ICInfA.2013.6720316>. 1416 1417 1418
 28. Perl, H.; Dechand, S.; et.al. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In Proceedings of the Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12–16, 2015; Ray, I.; Li, N.; Kruegel, C., Eds. ACM, 2015, pp. 426–437. <https://doi.org/10.1145/2810103.2813604>. 1419 1420 1421
 29. Harer, J.A.; Kim, L.Y.; et.al. Automated software vulnerability detection with machine learning. *CoRR* **2018**, *abs/1803.04497*, [1803.04497]. 1422 1423
 30. Russell, R.L.; Kim, L.Y.; et.al. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In Proceedings of the 17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17–20, 2018; Wani, M.A.; Kantardzic, M.M.; Mouchaweh, M.S.; Gama, J.; Lughofer, E., Eds. IEEE, 2018, pp. 757–762. <https://doi.org/10.1109/ICMLA.2018.00120>. 1424 1425 1426 1427
 31. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2022**, *19*, 2244–2258. <https://doi.org/10.1109/TDSC.2021.3051525>. 1428 1429
 32. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18–21, 2018. The Internet Society, 2018. 1430 1431 1432
 33. Li, Z.; Zou, D.; Tang, J.; Zhang, Z.; Sun, M.; Jin, H. A Comparative Study of Deep Learning-Based Vulnerability Detection System. *IEEE Access* **2019**, *7*, 103184–103197. <https://doi.org/10.1109/ACCESS.2019.2930578>. 1433 1434
 34. Pereira, J.D.; Campos, J.R.; Vieira, M. An Exploratory Study on Machine Learning to Combine Security Vulnerability Alerts from Static Analysis Tools. In Proceedings of the 9th Latin-American Symposium on Dependable Computing, LADC 2019, Natal, Brazil, November 19–21, 2019. IEEE, 2019, pp. 1–10. <https://doi.org/10.1109/LADC48089.2019.8995685>. 1435 1436 1437
 35. Li, X.; Wang, L.; Xin, Y.; Yang, Y.; Chen, Y. Automated Vulnerability Detection in Source Code Using Minimum Intermediate Representation Learning. *Applied Sciences* **2020**, *10*. <https://doi.org/10.3390/app10051692>. 1438 1439
 36. Srikant, S.; Lesimple, N.; O'Reilly, U. Dependency-Based Neural Representations for Classifying Lines of Programs. *CoRR* **2020**, *abs/2004.10166*, [2004.10166]. 1440 1441
 37. Zou, D.; Wang, S.; Xu, S.; Li, Z.; Jin, H. μ VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing* **2021**, *18*, 2224–2236. <https://doi.org/10.1109/TDSC.2019.2942930>. 1442 1443
 38. Dam, H.K.; Tran, T.; Pham, T.; Ng, S.W.; Grundy, J.; Ghose, A. Automatic Feature Learning for Predicting Vulnerable Software Components. *IEEE Trans. Software Eng.* **2021**, *47*, 67–85. <https://doi.org/10.1109/TSE.2018.2881961>. 1444 1445
 39. Y, F.; S, H.; C, H.; R, W. TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology. *PLoS ONE* **2019**, *14*. <https://doi.org/10.1371/journal.pone.0225196>. 1446 1447
 40. X, Z.; Pang, J.; Yue, F.; et al. A new method of software vulnerability detection based on a quantum neural network. *Sci Rep* **2022**, *12*. <https://doi.org/10.1038/s41598-022-11227-3>. 1448 1449
 41. Chen, Z.; Kommrusch, S.; Tufano, M.; Pouchet, L.; Poshyvanyk, D.; Monperrus, M. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Trans. Software Eng.* **2021**, *47*, 1943–1959. <https://doi.org/10.1109/TSE.2019.2940179>. 1450 1451
 42. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to Represent Programs with Graphs. In Proceedings of the 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net, 2018. 1452 1453 1454
 43. Dam, H.K.; Pham, T.; Ng, S.W.; Tran, T.; Grundy, J.C.; Ghose, A.; Kim, T.; Kim, C. Lessons learned from using a deep tree-based model for software defect prediction in practice. In Proceedings of the Proceedings of the 16th International Conference on 1455 1456

- Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada; Storey, M.D.; Adams, B.; Haiduc, S., Eds. IEEE / ACM, 2019, pp. 46–57. <https://doi.org/10.1109/MSR.2019.00017>. 1457
44. Zhou, Y.; Liu, S.; Siow, J.K.; Du, X.; Liu, Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In Proceedings of the Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada; Wallach, H.M.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E.B.; Garnett, R., Eds., 2019, pp. 10197–10207. 1459
45. Rabheru, R.; Hanif, H.; Maffei, S. A Hybrid Graph Neural Network Approach for Detecting PHP Vulnerabilities. *CoRR* **2020**, *abs/2012.08835*, [2012.08835]. 1460
46. Cheng, X.; Wang, H.; Hua, J.; Zhang, M.; Xu, G.; Yi, L.; Sui, Y. Static Detection of Control-Flow-Related Vulnerabilities Using Graph Embedding. In Proceedings of the 24th International Conference on Engineering of Complex Computer Systems, ICECCS 2019, Guangzhou, China, November 10-13, 2019; Pang, J.; Sun, J., Eds. IEEE, 2019, pp. 41–50. <https://doi.org/10.1109/ICECCS.2019.00012>. 1461
47. Chakraborty, S.; Krishna, R.; Ding, Y.; Ray, B. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Trans. Software Eng.* **2022**, *48*, 3280–3296. <https://doi.org/10.1109/TSE.2021.3087402>. 1462
48. Cao, S.; Sun, X.; Bo, L.; Wei, Y.; Li, B. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Inf. Softw. Technol.* **2021**, *136*, 106576. <https://doi.org/10.1016/j.infsof.2021.106576>. 1463
49. Wang, H.; Ye, G.; Tang, Z.; Tan, S.H.; Huang, S.; Fang, D.; Feng, Y.; Bian, L.; Wang, Z. Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection. *IEEE Trans. Inf. Forensics Secur.* **2021**, *16*, 1943–1958. <https://doi.org/10.1109/TIFS.2020.3044773>. 1464
50. Feng, H.; Fu, X.; Sun, H.; Wang, H.; Zhang, Y. Efficient Vulnerability Detection based on abstract syntax tree and Deep Learning. In Proceedings of the 39th IEEE Conference on Computer Communications, INFOCOM Workshops 2020, Toronto, ON, Canada, July 6-9, 2020. IEEE, 2020, pp. 722–727. <https://doi.org/10.1109/INFOCOMWKSHPS50562.2020.9163061>. 1465
51. Wu, Y.; Lu, J.; Zhang, Y.; Jin, S. Vulnerability Detection in C/C++ Source Code With Graph Representation Learning. In Proceedings of the 11th IEEE Annual Computing and Communication Workshop and Conference, CCWC 2021, Las Vegas, NV, USA, January 27-30, 2021. IEEE, 2021, pp. 1519–1524. <https://doi.org/10.1109/CCWC51732.2021.9376145>. 1466
52. Zou, D.; Hu, Y.; Li, W.; Wu, Y.; Zhao, H.; Jin, H. mVulPreter: A Multi-Granularity Vulnerability Detection System With Interpretations. *IEEE Transactions on Dependable and Secure Computing* **2022**, pp. 1–12. <https://doi.org/10.1109/TDSC.2022.3199769>. 1467
53. Nguyen, V.; Nguyen, D.Q.; Nguyen, V.; Le, T.; Tran, Q.H.; Phung, D. ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection. In Proceedings of the 44th 2022 IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022. IEEE, 2022, pp. 178–182. <https://doi.org/10.1109/ICSE-Companion55297.2022.9793807>. 1468
54. Keller, P.; Kaboré, A.K.; Plein, L.; Klein, J.; Traon, Y.L.; Bissyandé, T.F. What You See is What it Means! Semantic Representation Learning of Code based on Visualization and Transfer Learning. *ACM Trans. Softw. Eng. Methodol.* **2022**, *31*, 31:1–31:34. <https://doi.org/10.1145/3485135>. 1469
55. Wu, Y.; Zou, D.; Dou, S.; Yang, W.; Xu, D.; Jin, H. VulCNN: An Image-inspired Scalable Vulnerability Detection System. In Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022. ACM, 2022, pp. 2365–2376. <https://doi.org/10.1145/3510003.3510229>. 1470
56. Tufano, M.; Watson, C.; Bavota, G.; Penta, M.D.; White, M.; Poshyvanyk, D. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In Proceedings of the Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018; Huchard, M.; Kästner, C.; Fraser, G., Eds. ACM, 2018, pp. 832–837. <https://doi.org/10.1145/3238147.3240732>. 1471
57. Santos, E.A.; Campbell, J.C.; Patel, D.; Hindle, A.; Amaral, J.N. Syntax and sensibility: Using language models to detect and correct syntax errors. In Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018; Oliveto, R.; Penta, M.D.; Shepherd, D.C., Eds. IEEE Computer Society, 2018, pp. 311–322. <https://doi.org/10.1109/SANER.2018.8330219>. 1472
58. Harer, J.; Ozdemir, O.; Lazovich, T.; Reale, C.P.; Russell, R.L.; Kim, L.Y.; Chin, P. Learning to Repair Software Vulnerabilities with Generative Adversarial Networks. In Proceedings of the Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada; Bengio, S.; Wallach, H.M.; Larochelle, H.; Grauman, K.; Cesa-Bianchi, N.; Garnett, R., Eds., 2018, pp. 7944–7954. 1473
59. Mohan, V.T.; Jannesari, A. Automatic Repair and Type Binding of Undeclared Variables using Neural Networks. *CoRR* **2019**, *abs/1907.06205*, [1907.06205]. 1474
60. Hata, H.; Shihab, E.; Neubig, G. Learning to Generate Corrective Patches using Neural Machine Translation. *CoRR* **2018**, *abs/1812.07170*, [1812.07170]. 1475
61. Hajipour, H.; Bhattacharyya, A.; Fritz, M. SampleFix: Learning to Correct Programs by Sampling Diverse Fixes. *CoRR* **2019**, *abs/1906.10502*, [1906.10502]. 1476
62. Vasic, M.; Kanade, A.; Maniatis, P.; Bieber, D.; Singh, R. Neural Program Repair by Jointly Learning to Localize and Repair. In Proceedings of the 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net, 2019. 1477

-
63. Guo, D.; Ren, S.; et.al. GraphCodeBERT: Pre-training Code Representations with Data Flow. In Proceedings of the 9th
International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net, 2021. 1516
64. Chen, Z.; Kommrusch, S.; Tufano, M.; Pouchet, L.; Poshyanyk, D.; Monperrus, M. SequenceR: Sequence-to-Sequence Learning
for End-to-End Program Repair. *IEEE Trans. Software Eng.* **2021**, *47*, 1943–1959. <https://doi.org/10.1109/TSE.2019.2940179>. 1517
65. Ma, S.; Thung, F.; Lo, D.; Sun, C.; Deng, R.H. VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples.
In Proceedings of the Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo,
Norway, September 11-15, 2017, Proceedings, Part II; Foley, S.N.; Gollmann, D.; Sneekenes, E., Eds. Springer, 2017, Vol. 10493,
Lecture Notes in Computer Science, pp. 229–246. https://doi.org/10.1007/978-3-319-66399-9_13. 1518
66. Chakraborty, S.; Ding, Y.; Allamanis, M.; Ray, B. CODIT: Code Editing With Tree-Based Neural Models. *IEEE Trans. Software Eng.*
2022, *48*, 1385–1399. <https://doi.org/10.1109/TSE.2020.3020502>. 1519
67. Wu, L.; Li, F.; Wu, Y.; Zheng, T. GGF: A Graph-based Method for Programming Language Syntax Error Correction. In Proceedings
of the ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020. ACM,
2020, pp. 139–148. <https://doi.org/10.1145/3387904.3389252>. 1520
68. Yasunaga, M.; Liang, P. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback. In Proceedings of the
Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event. PMLR, 2020,
Vol. 119, *Proceedings of Machine Learning Research*, pp. 10799–10808. 1521
- 1522
- 1523
- 1524
- 1525
- 1526
- 1527
- 1528
- 1529
- 1530
- 1531