Empirical Software Engineering (2022) 27:2
https://doi.org/10.1007/s10664-021-10029-x

Deep security analysis of program code

A systematic literature review

Tim Sonnekalb[1] · Thomas S. Heinze[1] · Patrick Mäder[2,3]

Accepted: 27 July 2021Published online: 21 October 2021

Abstract
Due to the continuous digitalization of our society, distributed and web-based applications become omnipresent and making them more secure gains paramount relevance. Deep learning (DL) and its representation learning approach are increasingly been proposed for program code analysis potentially providing a powerful means in making software systems less vulnerable. This systematic literature review (SLR) is aiming for a thorough analysis and comparison of 32 primary studies on DL-based vulnerability analysis of program code. We found a rich variety of proposed analysis approaches, code embeddings and network topologies. We discuss these techniques and alternatives in detail. By compiling commonalities and differences in the approaches, we identify the current state of research in this area and discuss future directions. We also provide an overview of publicly available datasets in order to foster a stronger benchmarking of approaches. This SLR provides an overview and starting point for researchers interested in deep vulnerability analysis on program code.

Keywords Code inspection · Software security · Vulnerability detection · Deep learning · Supervised learning

Communicated by: Paolo Tonella

✉ Tim Sonnekalb
tim.sonnekalb@dlr.de

Thomas S. Heinze
thomas.heinze@dlr.de

Patrick Mäder
patrick.maeder@tu-ilmenau.de

[1] German Aerospace Center (DLR), Institute of Data Science, Jena, Germany

[2] Institute for Computer and Systems Engineering, Technische Universität Ilmenau, Ilmenau, Germany

[3] Faculty of Biological Sciences, Friedrich Schiller University, Jena, Germany

1 Introduction

With the continuous digitalization of our society, an increasing number of software and IT systems is used every day. Known vulnerabilities defined as "weakness in an information system, system security procedures, internal controls, or implementation that could be exploited by a threat source" (National Institute of Standards and Technology 2020) are constantly increasing. To illustrate the problem, the number of recorded vulnerabilities in the NIST National Vulnerability Database grew from 14,500 records in 2017 to 17,300 records in 2019 (National Vulnerability Database 2020). As distributed and web-based applications are omnipresent in many areas today, making them more secure gains paramount relevance. Furthermore, addressing security, by preventing and fixing vulnerabilities, early in a development process saves high costs analogous to failure prevention and fixing in general, which associated costs substantially rise in later development stages (Kumar and Yadav 2017).

Machine learning and especially deep learning (DL) methods gained importance in many research and application domains. In software engineering, e.g., deep learning is used for analysis and prediction based on software development artifacts, such as commits, issues, documentation, description, and, of course, program code. Thereby, program code can be either compiled object code or source code as written in a programming language. Models can be trained for different purposes, such as approximate type inference, code completion, and bug localization (Allamanis et al. 2018). One particular application of deep learning on program code is vulnerability analysis. The analysis is typically realized as binary classification, distinguishing vulnerable from non-vulnerable program code, or as a multi-class classification, additionally distinguishing the type of a vulnerability. These types typically follow the CWE (Common Weakness Enumeration) categorization system by The MITRE Corporation (2020).

This systematic literature review explicitly focuses on vulnerability analysis using deep learning approaches by detecting patterns in source code or object code. We identify a set of 32 relevant primary studies proposing deep vulnerability analysis of source and object code. The aim of the proposed methods is detecting known types of vulnerabilities in unseen program code rather than discovering new types of vulnerabilities. We briefly discuss the evolution from shallow networks to deep learning for this application. Further, we compare and contrast proposed methods considering a typical deep learning pipeline, ranging from data gathering, pre-processing, learning to evaluation. Finally, we provide rich discussions on code embeddings, network topologies, available datasets, and future trends in this area. Our results are relevant for researchers in security and software engineering, supporting them in finding new research directions and in conducting their ongoing research. The systematic and concise overview of deep learning approaches to vulnerability analysis on program code will also be helpful for beginners in this research area, as they can use our analysis as a guide in this complex and diverse field and in the tremendously growing list of machine learning literature.

The remaining sections of the paper are organized as follows: Section 3 provides a general overview of deep learning on code. Section 4 introduces our research questions and the methodology of this systematic review. In Section 5 we present and discuss findings per research question. We discuss trends and future directions in Section 6 and consider threats to validity in Section 7. Finally, Section 8 concludes the survey.

2 Related Work

Surveys on similar topics can already be found in the literature, and we will discuss them below. We would like to emphasize that in this work we distance from older, existing studies that use machine learning methods or code metrics, but not deep networks.

The survey by Allamanis et al. (2018) compares models for the domain of machine learning on code with various applications and discusses the naturalness of code. One application, he investigates is bug or code defect detection. The security of software could suffer as the number of bugs increases, but vulnerability detection in particular are the security-related bugs, which an attacker can exploit. The authors refer to deep learning methods as third wave of machine learning and consider them future work. The survey by Ucci et al. (2019) categorizes malware analysis methods by the use of machine learning techniques. Our work also covers the analysis of object code but especially vulnerabilities written by an engineer rather than detecting malicious code infiltrated by an attacker. Besides supervised learning methods, they also include unsupervised learning, which is common for abnormal behavior or code like malware. Lin et al. (2020) published a similar work on source code based vulnerability detection by deep learning. Our work covers partly the same primary studies, but our work discusses in addition the analysis of object code and the differences of the underlying deep learning architecture while this work focuses on deep neural networks. Computer security issues investigated by deep learning is the topic of the work by Choi et al. (2020). Their range of covered topics is much broader. Program analysis is one sub-area, which is discussed in this work in more detail. Similar to this work, Berman et al. (2019) consider deep learning techniques for the whole cyber-security domain as application. They have a much broader view than software. Even more broader is the selection of (Guan et al. 2018) with a spectrum of technical security issues. Ferrag et al. (2020) looks into network attack scenarios with deep learning for intrusion detection. Network security and software security are different fields with unequal requirements to the analysis. The survey by Ghaffarian and Shahriari (2017) focuses on vulnerability detection similar to our survey, but at the time of publication, deep learning was not yet used for this application and the author considers it a future direction. Nevertheless, we recommend new researchers in this field to read this publication because it covers anomaly detection and software metrics more in detail than we do. Similarly, Jie et al. (2016)'s survey reviews publications based on traditional machine learning methods.

3 Deep Vulnerability Analysis on Code

Deep learning (DL) is a subarea of machine learning, specifically concerned with the analysis of complex data using multi-layer neural network topologies. DL algorithms are suitable for supervised as well as unsupervised learning tasks and have been demonstrated useful for the analysis of program code. Current applications on program code are various, including malware identification via anomaly detection (Le et al. 2018; Cakir and Dogdu 2018), prediction of method names and types (Alon et al. 2019; Hellendoorn et al. 2018), semantic code search (Cambronero et al. 2019), and classification of vulnerable program code, which is the focus of this survey. While traditional machine learning employs manually crafted features, created in a step known as feature engineering, DL advances the machine learning concept towards representation learning, i.e., automatically extracting and learning features from the raw input data (LeCun et al. 2015). Given sufficient and representative training data, this methodological advancement allows for superior analysis results and removes the dependence on subject matter experts for defining features in often rather subjective processes. For text analysis, including program code, feature engineering has been non-trivial making DL an especially welcome technique (Gao et al. 2018 , [S1]).

Vulnerabilities are a subset of all software defects that may exist in a program code (Shin and Williams 2013) and the focus of deep vulnerability analysis is not finding new types of vulnerabilities but rather detecting a known type in new and unseen program code. To illustrate this explanation and our discussion in the later sections, we introduce an example of vulnerable source code suffering from an integer overflow (cp. Listing 1). The source code, written in C, operates based on a user input provided via a command line argument. This argument is being converted into an unsigned long integer (32bit) using the `strtoul()` function. The resulting value is passed on to a `test()`-function expecting an unsigned integer (16bit) as input. Inputs larger than the 16bit range, e.g., $n > 1,073,741,823$, result in an integer overflow on the three highlighted positions in the source code. An integer overflow is categorized as vulnerability type CWE-190 and may lead to overwriting of the stack if the size of the buffer is allocated smaller than the amount of data copied into it (cp. line 6 of Listing 1). The CWE aggregates vulnerabilities to classes and sub-classes, e.g., buffer access with incorrect length value (CWE-805) being a subcategory of improper restriction of operations within the bounds of a memory buffer or buffer overflow (CWE-119). In addition, CVEs (Common Vulnerabilities and Exposures) describe where an instance of a CWE has been discovered, e.g., CVE-2017-1000121 reports an integer overflow (CWE-190) found in Webkit. Such a vulnerability as in this example is hard to find for a developer by hand. A traditional static application security testing method (SAST) can hardly find overflows since there is no way to check automatically whether the calculation was performed correctly. A study by Russell et al. (2018) ([S2]) shows higher detection rates for a deep learning based approach in comparison to the SAST tools Clang, Flawfinder and CppCheck.

Traditional program analysis for finding vulnerabilities is based upon logical rule-based inference systems and heuristics. While having advantages, e.g., strong analysis soundness or completeness guarantees, these methods usually suffer from the undecidability of the underlying analysis problems and approach this problem using approximations of program behavior and sophisticated heuristics. Machine learning and in particular DL provides an orthogonal approach by focusing on statistical properties of software under the assumption that programs are written by humans and therefore follow regular patterns and code idioms (Pradel and Chandra 2021) (cp. naturalness hypothesis (Hindle et al. 2012; Allamanis et al. 2018)). In this way, DL methods for vulnerability analysis can better handle the often fuzzy patterns of software vulnerabilities and integrate natural text, such as comments and identifier names, into the analysis, which is usually omitted in traditional program analysis (Pradel and Chandra 2021). Furthermore, considering their statistical nature, DL methods also promise to cope better with the not well-defined characteristics of software vulnerabilities, which becomes apparent when considering the often reported high numbers of false positives for traditional program analysis (Johnson et al. 2013; Christakis and Bird 2016). The typical training pipeline of a DL model consists of four major phases: data gathering, pre-processing, learning and evaluation (cp. Fig. 1). Below, we provide a brief overview of each of these phases.

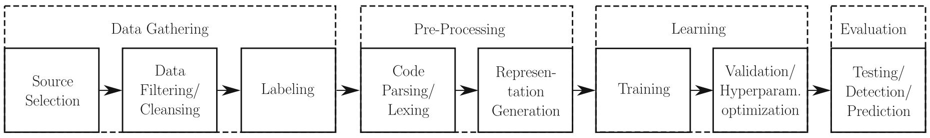
Data Gathering
Pre-Processing
Learning
Evaluation
Source Selection
Data Filtering/ Cleansing
Labeling
Code Parsing/ Lexing
Represen-tation Generation
Training
Validation/ Hyperparam. optimization
Testing/ Detection/ Prediction

Fig. 1 A typical deep learning pipeline for vulnerability analysis

```
#include <stdio.h>
#include <stdlib.h>

void test(unsigned int n) {
  int *buf, i;
  buf = malloc(n * sizeof *buf);     /*vulnerable*/
  if(!buf) return;
  for(i = 0; i < n; i++)
    buf[i] = i;                      /*vulnerable*/
  while(i-- > 0)
    printf("%x ", buf[i]);           /*vulnerable*/
    printf("\n");
  free(buf);
}

int main(int argc, char **argv) {
  int n;
  if(argc != 2) return 1;
  n = strtoul(argv[1], 0, 10);
  test(n);
  return 0;
}
```

Listing 1 Source code example into-bad1.c of Zou et al.'s dataset [S14]

Data Gathering Phase Essential to the prediction quality of a DL model is the availability of rich and representative training data. Supervised training methods additionally require high-quality labeled training samples in large quantities. Fortunately, software forges like GitHub[1] are an unprecedented source of data exploitable for DL on program code due to their increasing popularity and widespread usage for collaborative software development. However, project selection and data filtering play a vital role in separating large numbers of redundant and toy projects from projects containing production code and being well suited for training (Lopes et al. 2017; Kalliamvakou et al. 2014). A difficulty in using real program code is the typically large imbalance of vulnerable vs. non-vulnerable code in software projects. Therefore, often also synthetic program code is used to overcome the limited availability of labeled code samples and their strong imbalance. The most costly step in preparing a training set is labeling. Many vulnerability analysis approaches therefore employ labels produced by a static code analysis tool. As an alternative to assembling training data from scratch, already existing datasets can be used.

Pre-Processing Phase In the pre-processing phase, code samples are prepared and potentially annotated for the subsequent learning phase (cp. Fig. 1). While a major benefit of DL is representation learning, program code analysis still typically includes a certain amount of feature engineering, e.g., by annotating type information along the code. Extracted code samples are translated into single token statements or graphs by means of a specific parser or lexer for the respective source or object code. The generated tokens or graphs are further processed into numeric vectors by means of an encoding or embedding.

Learning Phase In this phase, the pre-processed training data is used to train a DL model towards performing a vulnerability analysis of program code (cp. Fig. 1). Typically, a binary

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

vulnerable vs. non-vulnerable or multi vulnerability type classification is being trained. Deep neural networks and their training process do not only consist of parameters automatically optimized in the training procedure, but also of many hyper-parameters with substantial impact on the performance of the resulting model. A validation step with careful hyper-parameter optimization is essential for training well-performing models (Komer et al. 2014). To ensure independent training, validation and evaluation results, training datasets are typically split into three respective parts. The majority of the dataset should be used for training with a typical proportion of the three splits being, e.g., 80:10:10.

Evaluation Phase Once training has converged, the resulting model will be evaluated with the independent evaluation split of the dataset or with additional datasets and benchmarks.

4 Methods

In order to identify a set of relevant primary studies to answer our research questions, we apply Kitchenham and Charters' four step guidelines for conducting reviews in software engineering: (1) defining research questions and selection criteria, (2) carrying out a comprehensive and exhaustive search for primary studies according to the criteria, (3) extracting data from the primary studies, and (4) answering the research questions with the gained data in a suitable presentation.

4.1 Research Questions

We conduct a systematic literature review on published research in the field of vulnerability analysis of program code using DL methods and thereby aim to answer the following six research questions:

RQ1 **Data demographics: How has machine learning based vulnerability analysis evolved over time?**
Motivation: This question aims to overview the development of this research field including a timeline of presented concepts and a categorization of primary studies.

RQ2 **Training data: How are training sets constructed and how are they structured?**
Motivation: This question aims to study the utilized training data in detail, their sizes, their class balance, and their methods of construction.

RQ3 **Code representation and encoding: How is program code made accessible to machine learning models for training?**
Motivation: This question aims to study program code pre-processing, representations and encodings, also with respect to granularity and arrangement of code samples.

RQ4 **Proposed and studied models: Which neural network topologies are used and how do they differ?**
Motivation: This question aims to emphasize the details of the utilized DL topologies and to compare their advantages and disadvantages.

RQ5 **Evaluation of proposed models: How are proposed models being evaluated?**
Motivation: This question aims to discuss and compare the evaluation of proposed approaches, their performance and the suitability of utilized metrics for evaluation.

RQ6 **Model generalizability: Are proposed methods generalizable to new and unseen projects?**

Motivation: This question aims to discuss the ability to transfer trained models to other software projects and to reuse trained models when training a model for a new problem.

4.2 Study Selection Process

For the search process, we use databases of computer science publishers and synoptic search engines (cp. Table 1). In a first step, we queried a combined pattern of search terms `S1 AND S2 AND S3 AND S4` across the databases, where

`S1` = (vulnerabilit* `OR` security);
`S2` = (analysis `OR` assessment `OR` detection `OR` discovery `OR` identification `OR` prediction);
`S3` = ("deep learning" `OR` "machine learning" `OR` supervised); and
`S4` = (code `OR` "byte code" `OR` "program code" `OR` "object code" `OR` "source code" `OR` software).

For databases 2–5, we were not able to query the entire search pattern at once since the search engine did not support all necessary operators. In these cases, we constructed individual queries containing only one term per group S1–S4 each and subsequently concatenated results. Table 1 shows the total number of retrieved results per database. In a second step, we filtered retrieved publications based on title and abstract applying a set of selection criteria (cp. Table 2). Retrieved results were sorted by databases' own relevance criterion and we terminated the search after 20 successive non-relevant publications due to the high number of results for databases 2–5. In a third step, we carefully read all remaining publications, evaluated our selection criteria again and only accepted primary studies not already in our set. Eventually, we performed an iterative snowballing search through each studies' references as listed in the publication, and citations retrieved through Google Scholar. With this procedure, we retrieved an additional four primary studies resulting in a total of 32.

Table 1 Queried databases and search engines

	Database	— Results ⟶		
		Retrieved	Filtered	Accepted
1	IEEE Xplore®	505	31	18
2	ACM Digital Library	4,325	23	4
3	Elsevier ScienceDirect®	9,229	27	3
4	Springer Digital Library	4,182	7	0
5	Google Scholar	≈ 12,000	39	2
6	Thomson Reuters Web of Science™	540	6	0
7	Scopus	177	10	1
	Snowballing	–	–	4
	Total	≈ 31,000	143	32

Table 2 Applied selection criteria for primary studies

A publication must be ...
– published in a journal, conference proceeding or a repository for computer science publications,
– written in English,
– proposing a deep learning approach,
– analyzing object or source code, and
– focusing on pattern-based vulnerability analysis.

4.3 Dataset Collection

We also collected datasets useful for researchers that want to evaluate their own vulnerability analysis approaches. We mainly identified these datasets through manual search. We started with the referenced and released datasets across the primary studies. Furthermore, we performed a manual search on Google Scholar, GitHub, and GitLab. We focused on datasets that either directly contain program code or link program code in an external repository and are accompanied by labels referring to vulnerable program code parts. If a primary study did not directly link a utilized dataset or a given link was broken, we manually searched for the download location or a mirror thereof.

5 Discussion

In order to get an overview of relevant keywords and their importance in the field of vulnerability analysis of program code we created a word cloud across all selected primary studies (cp. Fig. 2). The word cloud reflects the most prominent terms, i.e., *vulnerability*, *deep learning*, and *(source) code*, also reflecting and confirming our selection criteria. Furthermore, it shows that a *function* is an important concepts for analysis of code and that *neural network*, *features*, and *data (source)* are especially relevant terms when applying deep learning methods. In the following subsections, we discuss the results that we retrieved from the primary studies in order to answer the research questions outlined in Section 4.1.

5.1 RQ1: Data Demographics

The application of deep learning methods has substantially increased across many research disciplines in the last years. We observe a similar trend also for vulnerability analysis of program code (cp. Fig. 3). Figure 3 presents a timeline depicting the evolution of vulnerability analysis approaches that employ machine learning. We connect primary studies with an arrow if the latter references the former and proposes its improvement. The progression in this figure can be distinguished into three stages based on the proposed methods: traditional machine learning methods (shaded white), shallow learning neural networks (shaded light orange), and deep learning neural networks (shaded dark orange). Initial studies proposing traditional machine learning for vulnerability analysis were published in 2014, while first approaches proposing deep learning were published in 2017. Current research almost entirely focuses on deep learning approaches, as the increase in studies per year shows. This trend is accompanied by increasing complexity of the software as well as size of the software projects since DL can better process complex information. Traditional machine learning has

Fig. 2 Word cloud generated from the 32 primary studies

often only one processing layer, which cannot abstract meaningful patterns from the underlying complex data. The advantage of DL are successive abstraction layers for filtering important information and classifying the code snippet to the correct category. We further observe that authors who beforehand proposed traditional machine learning techniques later proposed DL methods. Note that this is not a complete overview of traditional machine learning-based work done in this area. We refer the reader to related surveys mentioned in Section 2.

Answer to RQ1: We observe a clear trend towards deep learning approaches often motivated by the complex nature of program code structure.

5.2 RQ2: Training Data

The training of a deep neural model to perform a classification task requires a substantial amount of data representing the classes to be distinguished. Deep neural networks not only

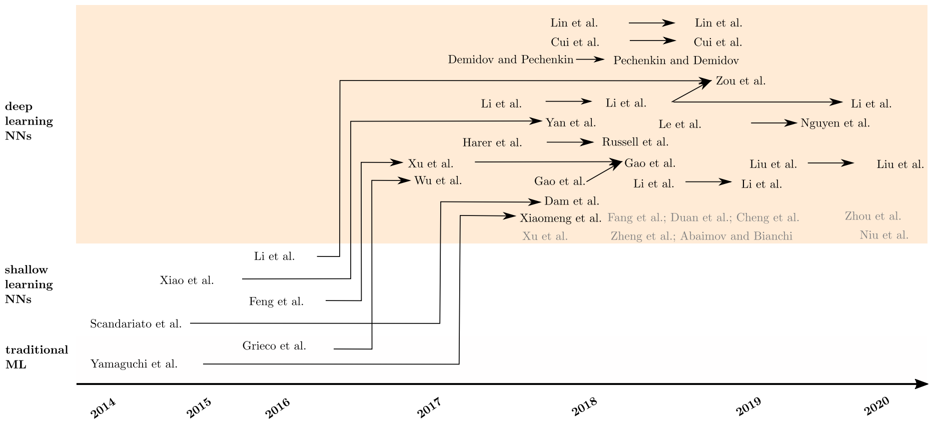

Fig. 3 Evolution of approaches over time. Studies with an incoming arrow explicitly state that they build on the connected previous work or build upon their own previous work

train network layers performing the final classification task, but also a cascade of additional layers that extract the most relevant features for performing this task. This approach has been demonstrated to be superior for many tasks, but also means that the selection of suitable training data is crucial. A relevant factor is the representativeness of training data for the task to be solved. Below, we discuss individual attributes of training data to be considered when performing vulnerability analysis.

Size and composition of the dataset In order to not only train a shallow classifier, but also a cascade of problem-specific feature extractors, deep neural networks require substantially more training data. That means the size of the training dataset influences the performance of a trained model. Especially when training deep models on source code, a large number of training examples are needed to generalize from developer-specific code idioms. Furthermore, a dataset should be representative for the intended application domains (Dam et al. 2018, [S3]), e.g., operating systems or web applications, and accordingly combine vulnerability samples across the different domains. Figure 4 shows a histogram of dataset sizes used across the primary studies. We found that sizes greatly vary from under 10k to more than 1M samples per study. While we observe a trend towards larger studies for DL on source code,

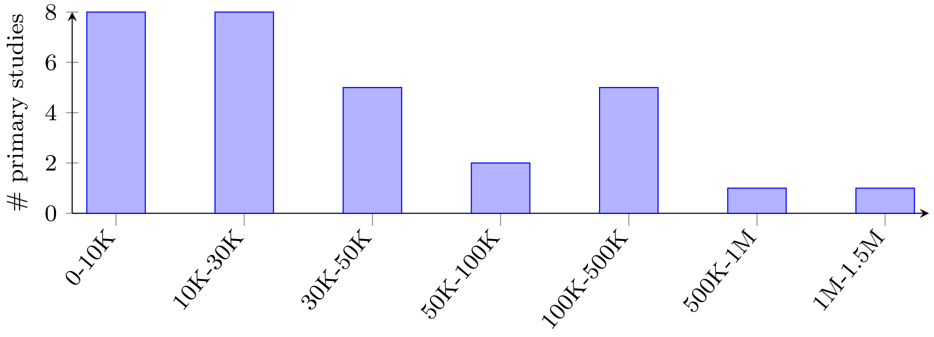

Fig. 4 Number of samples in the evaluated datasets (30 primary studies report dataset size)

e.g., compare the frequently used public Github archive dataset containing around 182,000 software projects amounting to a size of 3 TB (Markovtsev and Long 2018), this trend is not visible for vulnerability analysis yet (cp. Fig. 4).The largest available dataset contains roughly 1M samples (Russell et al. 2018, [S2]). For the training, validation and testing procedures, these datasets are split into shares, with primary studies reporting the following splitting ratios: 8:1:1, 7:1:2, and 6:2:2 (cp. Section 5.5).

Synthetic training data An alternative to real training data is the generation of synthetic code. A potential benefit is the chance to generate representatives of vulnerabilities that only rarely occur and are therefore hard to find in reality. For example, Pechenkin et al. (2018) ([S4]) use a random generator, but do not publish their implementation. In fact, we did not find a single synthetic code generator for vulnerabilities that would be publicly available or even open-sourced. Studies show that there remains a gap between synthetic samples and real-world samples. Russell et al. (2018) ([S2]) found that precision substantially dropped when they analyzed real test samples with a model solely trained with synthetic samples. A potential strategy can be mixing real and synthetic samples to gain the benefits of a larger, balanced training set without sacrificing on representativeness. Li et al. (2018) ([S5]) and Zou et al. (2019) ([S23]) applied this strategy successfully. Studies containing synthetic samples often use subsets of the SARD dataset (Li et al. (2021) [S7], Zheng et al. 2019) [S8]). The software assurance reference dataset (SARD) created and published by the NIST institute (Black 2018) contains synthetic code samples (cp. Table 4 in the Appendix).

Generating labels Today, deep vulnerability analysis is almost solely approached in a supervised manner, i.e., all primary studies use training sets accompanied by labels. A label annotates a code entity as belonging to one or multiple of the classes to be distinguished. For a binary classifier, these would be vulnerable and non-vulnerable, while a more sophisticated approach could, e.g., use enumerated weaknesses (CWE) as classes Niu et al. 2020([S21]). In contrast, unsupervised optimization would train a model to discover and cluster input data automatically according to attributes like vulnerability. While such an approach would have a fundamental benefit by operating without the need for labels, the variability of program code makes training a successful unsupervised approach hard and a future exercise. Labeling a dataset theoretically means that an expert reviews a given codebase and identifies vulnerable parts. This approach is unrealistic when aiming for a large dataset as required for the training of deep models. An alternative source of labels, often retrievable in an automated manner, are vulnerability collections, e.g., CVE reports in the (National Vulnerability Database 2020), and three primary studies apply this strategy. When discovered in an open-source project, listed CVEs can be traced back to the respective code location. Note that a CVE can be associated with a CWE, but this is not necessary and thus not always the case. By analyzing the code versioning system and issue tracker, further information, such as the vulnerability-fixing code change or the introduction of the vulnerability can also be extracted. Harer et al. propose another automated labeling approach by utilizing static code analysis tools (Harer et al., 2018, [S9]). However, static analysis will only discover what is captured by the applied rule set and a model trained on the analysis' results will essentially approximate this rule set. The authors argue that a pessimistic analysis aiming for high recall, could ease a later manual labeling by reducing the amount of code to be evaluated. Nguyen et al. (2019) ([S10]) propose a semi-automated approach by clustering code samples before labeling. Various authors argue that a better labeled dataset would facilitate higher accuracy of their proposed approach and aim to create these in the future (Russell et al. (2018) [S2], Li et al. (2018) [S5], Li et al. (2021) [S7]).

Scope of labels The granularity of a training sample must not necessarily be the same as the the scope of an associated label. Li et al. (2021) ([S7]) argue that the labeling should be as accurate as possible, optimally on line-of-code level, but automated labeling processes often do not provide this fine-grained information (see above).

Label balance An ideal training represents classes in a balanced manner to the model. In contrast, an imbalanced training typically has negative effects on a model's performance as the model implicitly learns the distribution of classes. When trying to distinguish very rare classes like vulnerable code from very dominant classes like non-vulnerable code, it can be a safe and easy to learn strategy for a model to always predict the dominant class while still reaching high accuracy. When considering vulnerability analysis on software projects, this imbalance is naturally given as software typically contains large parts that are non-vulnerable. Figure 5 shows the distribution of vulnerable and non-vulnerable training samples used by the 20 primary studies reporting this information. We observe highly varying proportions of 1% to 52% vulnerable samples per studied dataset. Thereby, 1% of known vulnerable code more realistically reflects the situation in software projects today, but constitutes a highly imbalanced datasets; while values beyond 40% constitute an almost balanced dataset better suited for training. We argue that active measures should be taken to reach a balanced training set. Simple strategies, long known from traditional machine learning, are over- and under-sampling. Over-sampling means that samples of the minority class appear more than once in the training set to counter-balance the dominant class. Three studies apply this strategy (Liu et al., 2020, [S11]). A drawback, however, is that over-sampling does not actually make the training set richer in terms of new and unseen samples. Under-sampling refers to reducing the samples of all classes to the amount of samples in the least represented class. This strategy is applied by Li et al. (2019) ([S12]). The drawback of this strategy is that the overall training set may become substantially smaller resulting in insufficient data for training a generalizing model not over-fitted to the training set. A trade-off is using both, under-sampling and over-sampling as used by Cui et al. (2018) ([S13]), Cui

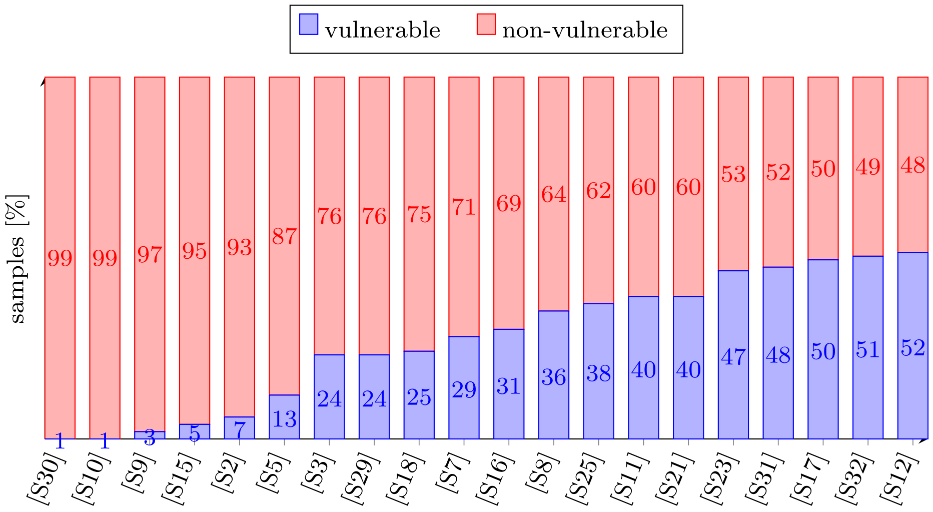

Fig. 5 Division of vulnerable and non-vulnerable samples across datasets of the primary studies (20 studies report the distribution)

et al. (2019) ([S14]). A variant proposed by Liu et al. (2019) ([S15]) is fuzzy oversampling meaning that existing data is enriched by synthetic examples through modifications of the feature vector. Another strategy, also in combination with the ones above, is utilizing an objective function that considers and weighs the imbalance during the training process and was applied by Fang et al. (2019) [S16], Lin et al. (2019) [S17], Russell et al. (2018) [S2].

Vulnerabilities A majority of 22 primary studies build their datasets upon known vulnerability categories, 19 studies utilize the CWE types of (The MITRE Corporation 2020) and 3 studies utilize CVE alarms (National Vulnerability Database 2020), while 10 studies use program code samples merely labeled as vulnerable or non-vulnerable. Figure 6 provides an overview across the most common utilized vulnerabilities. The common CWE categorization is divided into classes and sub-classes, so one CWE can be contained in another CWE. We found the most common analyzed CWE to be CWE-119 "Improper Restriction of Operations within the Bounds of a Memory Buffer", a parent category of the classic buffer overflow. Often, this CWE occurs in combination with CWE-399, the parent category of all resource management errors. Both vulnerability types are typically introduced through multiple statements, making a pattern-based detection an appropriate approach. Other investigated CWE-types are OS command injection (CWE-78), e.g., SQL injection, improper input validation (CWE-20), and NULL pointer dereference (CWE-476). We aggregate roughly 50 other CWE types that solely occur in a single study in Fig. 6. We also observed that authors not necessarily classify all vulnerability types available in their utilized datasets but sometimes rather chose to train a binary classifier distinguishing merely between vulnerable and non-vulnerable code.

Benchmark datasets Benchmark datasets are essential when comparing results of competing approaches and for researchers that want to develop new and improved approaches. Across the primary studies and through an additional search (cp. Section 4.3), we discovered 20 datasets publicly available for benchmarking (cp. Table 4). Most of the datasets provide samples as source code mostly written in one programming language (considering C and C++ as one language), except for SARD that mixes C/C++ and Java and VulinOSS that includes various languages. One dataset (No. 19), contains pre-processed object code as numeric vectors. Twelve datasets contain real source code collected from free/libre open source software (FOSS) projects, while three are solely comprised of synthetic samples and the remaining five are mixed. The synthetic examples are usually derived as a subset of the

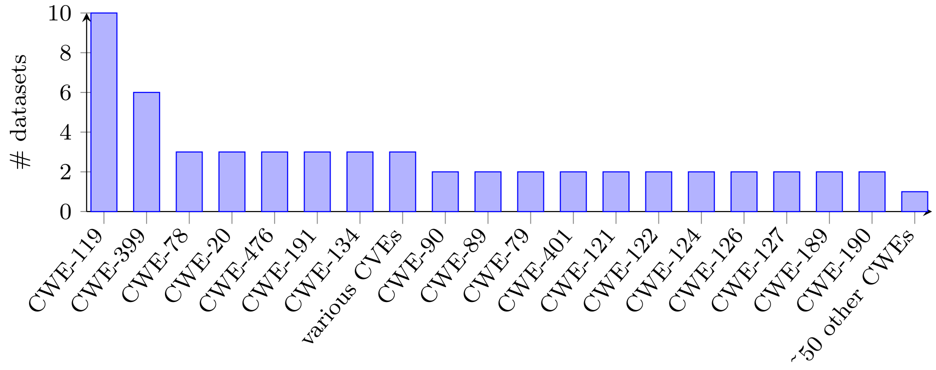

Fig. 6 Common vulnerabilities utilized in the datasets of the primary studies

SARD dataset or with methods like fuzzing from real-world examples. One sample in these datasets typically spans one method or one file. All datasets contain either binary labels (vulnerable vs. non-vulnerable) or multi-class labels typically corresponding to CWEs or CVEs annotating multiple tokes, a statement, a method or a whole file.

Answer to RQ2: Training sets are constructed from real program code as well as from synthetically generated code. Datasets are accompanied by labels created manually, derived through vulnerability collections or with the help of static code analysis tools. The naturally existing class imbalance should be taken into account during dataset construction and later in the DL pipeline.

5.3 RQ3: Code Representation and Encoding

With this research question we explore how primary studies represent and encode program code for making it suitable as input when training a deep model. Table 3 provides an overview across all 32 primary studies in terms of input data, program code representation and the trained model. When discussing this research question, we specifically refer to the table's first five columns.

Granularity of training samples When training on program code, a fundamental decision is the granularity in which to represent the code to the model. Granularity thereby refers to the extent of one training sample rather than the input granularity of a trained model which may be smaller, e.g, for sequentially trained recurrent networks. Typical granularities range from statement to file. In general, granularity should be chosen so that a code entity provides sufficient characteristic information for classifying it as potentially vulnerable or not. Thereby, an example for a vulnerability on statement level is an incorrect conversion between numeric types (CWE-681), while storing sensitive data in a mechanism without access control (CWE-921) can typically only be identified when analyzing multiple methods together. Given only this observation, one might consider coarse-grain code entities as universally suitable. However, there are also benefits in processing more fine-grained code entities. Splitting a code base into smaller entities results in more training data, which is typically beneficial for the performance of a deep model. Furthermore, if the vulnerable part is small and constitutes only a fraction of a code entity, it is harder to train a model towards identifying this small portion of characteristic information. The first column of Table 3 refers to the code granularity chosen by the primary studies. Primary studies operate: on single statements; on multiple statements, either consecutive statements or program slices; on single methods; or on file-level containing multiple methods. The table shows that studies often focus on multi-statement and method-level granularities and we observe a trend towards more fine-grained analysis in more recent studies potentially to provide users with more precise results (Li et al. (2020) [S18], Zou et al. (2019) [S23]). Source code functions by nature differ a lot in their size, so aggregating functions by combining several semantically related statements to an intermediate non-executable code can help to adjust the length of very long code fragments (Li et al. 2021, [S7]). An interesting approach is function inlining where method calls within one method are replaced by the code of the other function resulting in a wider scope of the vulnerability analysis (Liu et al. (2020) [S11], Li et al. (2019) [S19]). We observe correlations between the vulnerability type to be identified (cp. Fig. 6) and the sample granularity for analysis: SQL injections (CWE-89) are identifiable on statement-level granularity, while a buffer-overflow related vulnerability (CWE-119, CWE-121, CWE-122 etc.) typically assembles over multiple statements and at least requires this

Table 3 Characteristics of the 32 identified primary studies applying deep vulnerability analysis on program code

Granu	Code	Representation			Model			Ground	Stu	Ve	Arti
larity	format	Data	Serial.	Encoding	Topology	Outp.	Depth	truth	dy	nue	facts
Statement	C/C++, Python	method name	character sequence	one-hot	FC+BiLSTM, **LSTM+BiLSTM**	FC	3	CVE DB	[S12]	IEEE Access	code & data
	SQL	SQL query	manual encoded seq.	numeric	**CNN**	FC	1	given dataset	[S24]	IEEE Access	data
Multi-statement	C/C++	TPP	word frequencies seq.	word2vec CBOW	**CNN+BiLSTM**	FC	6	synthetic	[S21]	MEA	data
		CFG	word seq.	word2vec	**BiLSTM**	FC	2	CVE DB, synthetic	[S5]	NDSS	data
		AST, PDG	aggregated word seq.	one-hot	CNN, LSTM, GRU, BiLSTM, **BiGRU**	FC	2	CVE DB, synthetic	[S7]	TDSC	code & data
		AST, PDG	aggregated word seq.	one-hot	BiLSTM, **BiGRU**	Pool.	2	CVE DB, synthetic	[S18]	TDSC	code & data
		SDG	aggregated word seq.	word2vec skip-gram	**BiLSTM**	FC	3	CVE DB, synthetic	[S23]	TDSC	data
	Object Code	plain code	instructions	one-hot	**LSTM**	FC	2	emulation	[S20]	ICPS	–
		plain code	instruction components	one-hot	**BiLSTM**	FC	3	emulation	[S4]	SIN	–
		plain code	machine instruction seq.	one-hot	RNN, **SeqVAE**	FC	n.g.	given dataset	[S32]	ICLR	code & data
		plain code	basic blocks	one-hot	**CNN**	FC	3	given datasets	[S19]	IEEE Access	–

Table 3 (continued)

Granu larity	Code format	Representation Data	Serial.	Encoding	Model Topology	Outp.	Depth	Ground truth	Stu dy	Ve nue	Arti facts
		DFG	word sequence	word2vec	RNN, **BiRNN**, LSTM, BiLSTM, GRU, BiGRU	FC	3	synthetic	[S8]	ACAI	–
Method	C/C++	plain code	lexer seq.	random one-hot	**CNN**, RNN, GRU	RF	1	SAST	[S2]	ICMLA	data
		CFG	lexer seq.	word2vec	**CNN**	ET	2	SAST	[S9]	arXiv	–
		CPG	lexer seq.	word2vec	**CNN**, LSTM	n.g.	n.g.	synthetic	[S22]	ICAIT	–
		AST	DFT seq.	word2vec CBOW	**BiLSTM**	FC	2 1	synthetic, CVE DB	[S17], [S15]	TDSC, TFS	–
		plain code	instruction seq.	one-hot	**BiLSTM**	FC	1	clustering	[S10]	IJCNN	–
		plain code	instructions	one-hot	CNN, **BiLSTM**	FC	n.g.	CVE DB, given data	[S11]	TII	partly: data
		CFG	graph	Doc2vec	**GCN**	FC	1	synthetic	[S26]	ICE- CCS	–
		AST	DFT sequence	word2vec	**BiLSTM**	FC	1	manual	[S30]	TII	code & data
		combined graph	graph	word2vec	CNN, BiLSTM, **GGRU+CNN**	FC	2	manual	[S23]	NIPS	partly: data
		CPG	embedded graph tensor	binary (edge exis-tence)	**VAE**	FC	n.g.	synthetic	[S25]	IJCAI	–
	Java	AST	word seq.	codebook	**LSTM**	RF	1	given dataset	[S3]	TSE	–
	object code	CFG	manually defined	struc-ture2vec	**FC**	Sim.	n.g.	CVE DB	[S1]	ESEC/ FSE	–
File	PHP	plain code	generic token seq.	word2vec	CNN, RNN, **LSTM**, BiLSTM, GRU	FC	n.g.	synthetic	[S16]	PLOS ONE	–

Table 3 (continued)

Granu larity	Code format	Representation Data	Serial.	Encoding	Model Topology	Outp.	Depth	Ground truth	Stu dy	Ve nue	Arti facts
	C	program slices	word seq.	numeric	CNN, LSTM, **CNN+LSTM**	FC	2	synthetic	[S31]	ICSAI	–
	Java-Script	AST	calculated mea-sures	–	CNN, RNN, **CNN+LSTM**	FC	2	given data	[S29]	JSSO	–
	Object	plain code	bit-wise matrix	image pixel	**CNN**	FC	2	given dataset	[S13, S14]	TII, JPDC	–
	Code	plain code	instruction seq.	numeric	**CNN**, LSTM, CNN+LSTM	FC	1	fuzzing	[S28]	ICCC	–
		ACFG	graph	struc-ture2vec	**FC**	Sim.	2	given dataset	[S27]	CCS	–

Graphs: ACFG – Attributed Control Flow Graph, ACFG – Attributed Control Flow Graph, AST – Abstract Syntax Tree, CFG – Control Flow Graph, CPG – Code Property Graph (includes AST, CFG and PDG), DFG – Data Flow Graph, PDG – Program Dependence Tree, SDG – System Dependence Graph, TPP – taint propagation path

Topologies: BiGRU – Bidirectional Gated Recurrent Unit, BiLSTM – Bidirectional Long Short-Term Memory, BiRNN – Bidirectional Recurrent Neural Network, CNN – Convolutional Neural Network, FC – Fully Connected Neural Network, GCN – Graph Convolutional Networks, GGRU – Graph Gated Recurrent Unit, GRU – Gated Recurrent Unit, LSTM – Long Short-Term Memory, RNN – Recursive Neural Network, RCNN – Regions with Convolutional Neural Network, RF – Random Forest, RT – Random Trees Classifier, Pool – Global Max Pooling Layer, SeqVAE – Sequential Variational Auto-Encoder

Venues: TDSC – IEEE Transactions on Dependable and Secure Computing, ICPS – International Conference on Industrial Cyber-Physical Systems, SIN – International Conference on Security of Information and Networks, NDSS – Network & Distributed System Security Symposium, MEA – Journal Measurement, ICLR – International Conference on Learning Representations, ACAI – Advances in Computing and Artificial Intelligence, ICMLA – International Conference on Machine Learning and Applications, ICAIT – International Conference on Advanced Infocomm Technology, TFS – IEEE Transactions on Fuzzy Systems, IJCNN – International Joint Conference on Neural Networks, TII – Transactions on Industrial Informatics, ICECCS – International Conference on Engineering of Complex Computer Systems, NIPS – Conference on Neural Information Processing Systems, IJCAI – International Joint Conference on Artificial Intelligence, TSE –Transactions on Software Engineering, ESEC/FSE – European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ICSAI – International Conference on Systems and Informatics, JSSO – Journal of Systems and Software, JPDC – Journal of Parallel and Distributed Computing, ICCC – International Conference on Computational Creativity, CCS – Conference on Computer and Communications Security

Other: CVE DB – CVE database (National Vulnerability Database 2020), DFT – Depth First Traversal, ET – Extra Trees Classifier, n.g. – not given, Pool. – Pooling Layer, Sim. – similarity, SAST – Static Analysis Security Testing granularity for identification. One study (Li et al. 2021, [S7]) proposes differing input and output granularities. Their pipeline receives samples on program level as input and classifies on program slices.

Code format and language The second column "Code format" of Table 3 refers to whether a study analyzes object code or source code and lists the analyzed programming languages for primary studies analyzing source code. A majority of 22 primary studies analyze source code, stemming from code written in seven different programming languages. The most often analyzed programming languages are C and C++ possibly due to their wide usage for developing core functionality of connected systems. The languages' versatility and low-level features offer many possibilities for software developers but also allow introducing many vulnerabilities, accordingly C and C++ are considered among the most insecure programming languages (Michaud and Painchaud 2008). Although, other languages like Python and PHP should prevent some vulnerabilities on byte-operation level, they may allow other vulnerabilities and their libraries may also still be vulnerable, e.g., for buffer overflows (National Vulnerability Database 2014), due to the fact that they are, e.g., written in C. PHP, JavaScript and SQL are commonly used for developing web-applications and are vulnerable to code-injections, such as SQL injections (CWE-89). Li et al. (2019) ([S12]) aim to predict the vulnerability of methods written in C++ and Python. Since they solely analyze method names, a joint model for both programming languages could be trained. A potential reason for focussing primarily on source code for analysis is its richer representation containing, e.g., method names, variables names, comments that can be exploited, also in combination with other data sources, such as documentation and issue tracker information, for a vulnerability analysis. Harer et al. (2018) ([S9]) evaluated source code vs. object code for vulnerability analysis and their results in terms of higher ROC AUC and PR AUC for source code support this assumption (cp. Section 5.5). However, several primary studies focusing on source code confess that they aim to expand their work to object code, arguing that a vulnerability analysis may not only be conducted by developers but also by consumers of proprietary software that is not open-sourced (Li et al. (2018) [S5], Lin et al. (2019) [S17], Dam et al. (2018) [S3], Liu et al. (2019) [S15]). The processing of object code typically starts from binary files that are translated with a dis-compiler, such as IDA Pro or Capstone, into machine instructions or assembler code. Liu et al. (2020) ([S11]) compile the source code to object code and disassemble the object code to combine both views for analysis. While object code abstracts from the programming language, it is architecture-specific meaning that instructions can differ across hardware platforms, which potentially complicates analysis (Pechenkin et al. (2018) [S4], Demidov and Pechenkin et al. (2018) [S20]).

Intermediate code representation There exist multiple strategies for how to prepare program code for training of a model (cp. column "Representation" of Table 3). Primary studies use formats that range from the given plain code to more complex representations, e.g., abstract syntax tree (AST), control flow graph (CFG), program dependence graph (PDG), as known from compilers or static analysis tools. More recent approaches even introduce customized formats, such as the Attributed Control Flow Graph (ACFG), the Code Property Graph (CPG) or a graph combined of several of the former Xiaomeng and Pechenkin et al. (2018) ([S22]). The intermediate transformation of code into a graph representation is often proposed as code is not executed linearly like text but in branches and conditions preferably reflected in a graph representation. Which graph is most suitable depends on the type of vulnerability to be analyzed (Zhou et al. 2019, [S23]). For example, program or system dependence graphs represent control as well as data dependencies within code executions making it helpful to detect resource management vulnerabilities (e.g., CWE-399). More than half of the primary studies (19) use at least one graph representation or combine several graphs into one meta-graph, suggesting that treating code as graph or tree is suitable for deep vulnerability analysis. Also a combination of input granularities has been proposed in order to combine features of multiple abstractions. Dam et al. (2018) ([S3]) took the same feature set in a local pooling and a global clustering and combine both into a classifier, while Zhou et al. (2019) ([S23]) combine a node embedding and a graph embedding encoding one code granularity into the edges and the other into the nodes of the same graph. Adding semantic information during the pre-processing of code, e.g., by annotating token types, can improve analysis performance (Abaimov and Bianchi, 2019, [S24]), so that there is no need to learn this information from the training data. Including such additional knowledge is proposed as future work in various primary studies (Lin et al. (2019); Li et al. (2019), Li et al. (2019)).

Serialization The fourth column “Serial.” of Table 3 refers to the serialized features from the intermediate representation that are used as input to the deep model. For plain source code, this is often a word sequence created by a lexer, while object code is often disassembled into a sequence of instructions. Graphs are typically translated into paths, e.g., execution paths, or traversed by a depth first search (DFT). These features need to be further translated into numeric sequences that can be processed by a neural network. This step is called encoding.

Encoding program code A neural network expects a n-dimensional vector of numerical values representing a program code sample to be trained or analyzed. This necessitates a transformation of program code given in one representation, e.g., text or graph, into this numerical representation. Ideally, this transformation should preserve all relevant information for the given task, e.g., similar tokens are assigned similar numerical representations (Chen and Monperrus 2019). There are various ways for encoding program code (cp. column “Encoding” of Table 3). A simple approach is building a dictionary and enumerating each token to be encoded in a sparse vector of binary values containing a single one at the position of the given token. This form of encoding is called one-hot encoding and is applied by ten primary studies. A numerical encoding is similar to that and assigns every word a consecutive number. Cui et al. (2019) ([S14]) propose a more exotic form of encoding by reading tokens of object code bit-wise and representing 8 bits as a pixel in a grayscale image, thereby, gaining the ability to use a standard convolutional neural network as frequently applied for computer vision tasks. Li et al. (2019) ([S19]) compared a numeric encoding and a one-hot encoding and found the latter to yield higher accuracy at the cost of an increased training time.

Word embeddings A more advanced encoding is an embedding, e.g., learning a transformation of sparse natural language data into a dense, low-dimensional representation. In recent years, several successful embeddings for text, e.g., word2vec (Mikolov et al. 2013), Glove (Pennington et al. 2014), and Fasttext (Bojanowski et al. 2017), have been proposed. The most wide-spread of these embeddings is word2vec (Mikolov et al. 2013), which predicts a word based on its context words (CBOW) or a context word based on the target word (skip-gram). Nine primary studies use word2vec, but not all specify whether they used the CBOW or the skip-gram version. Important parameters of an embedding are vocabulary size and the dimension of the embedding. Unfortunately, only five primary studies mention these parameters. Duan et al. (2019) ([S25]) found that a vocabulary size of 128 and an embedding dimension of 144 worked best for them.

Graph embeddings Word embeddings, such as word2vec, only embed tokens and thereby neglect known dependencies among them. In contrast, graph embeddings aim to encode tokens as well as their dependencies into a dense, low-dimensional representation. Given the graphical nature of program code, graph embeddings appear to be a more suitable encoding. Five primary studies propose graph embeddings: the Graph-GRU (GGRU) studied by Zhou et al. (2019) ([S23]), the graph convolutional network (GCN) (Kipf and Welling 2017) studied by Cheng et al. (2019) ([S26]), and the structure2vec framework proposed by Dai et al. (2016) and studied for vulnerability analysis by Xu et al. (2017) ([S27]) and Gao et al. (2018) ([S1]). Additionally, Duan et al. (2019) ([S25]) propose the Doc2vec embedding, which is based on word2vec and was originally developed to embed whole documents. The authors use Doc2vec to encode a CFG's nodes, representing tokens, and additionally encode a CFG's edges into a feature tensor. Cheng et al. (2019) ([S26]) compared their GCN graph embedding with a token-based one-hot encoding on a dataset with labeled business logic errors (CWE-840). They found that the graph embedding outperformed the token-based encoding in terms of F-measure and area under the ROC.

Answer to RQ3: In the most simple form, program code is separated into tokens that are then encoded as numerical values. A more advanced form is training an embedding transforming sparse program code tokens into dense, low-dimensional vectors. However, common for sophisticated code pre-processing is an intermediate graph-based representation, utilizing diverse graphs dependent on the vulnerabilities to be analyzed. Graphs are either directly encoded or transformed into a token sequence or matrix.

5.4 RQ4: Proposed and Studied Models

The aim of this research question is exploring the model topologies proposed by the different primary studies and their characteristics (cp. column "Model topology" of Table 3). Figure 8 illustrates the distribution of network topologies proposed by the primary studies with quantities referring to the usage across the typically multiple performed experiments per study.

Extent of vulnerability analysis Figure 7 provides an overview of the attempted vulnerability analysis per primary study in terms of vulnerability classes to be distinguished. A majority of 28 primary studies perform binary classification, e.g., differentiating vulnerable from non-vulnerable program code samples. Merely four primary studies aim to distinguish multiple vulnerability types and select 5 to 40 classes to be identified and differentiated.

Recurrent topologies Primary studies predominantly propose recurrent neural networks (RNN) for performing vulnerability analysis of program code (cp. red pie in Fig. 8). Recurrent network topologies maintain a memory of previously seen input data by incorporating the previous state of hidden units when computing an updated state. This property makes recurrent topologies especially suitable for sequential data of variable input length as given when analyzing program code. RNNs have been first proposed by Hopfield (1982) and since been evolved substantially. A main driver in this evolution is a problem called vanishing

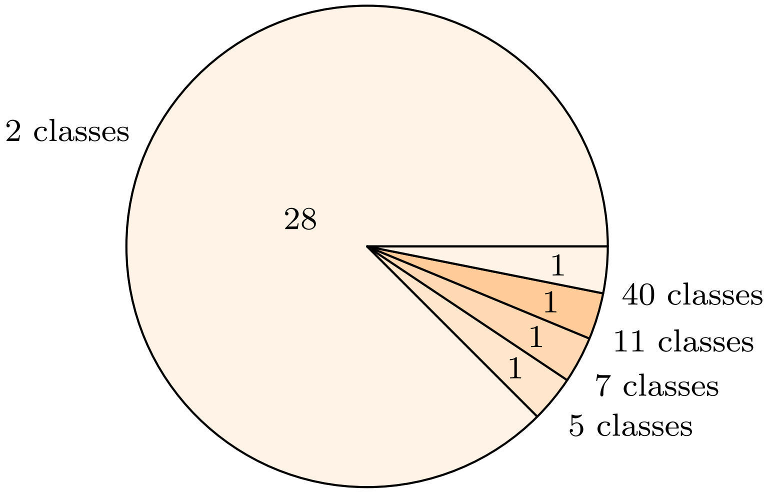

Fig. 7 Extend of the evaluated classification problem across primary studies

gradients, existing for all deep neural networks but becoming especially relevant for recurrent topologies exposed to long sequences of input data. A major improvement in mitigating the vanishing gradient problem are gated recurrent topologies that allow the model to learn and control which previous information shall be maintained and which other information can be discarded. Gated topologies appear in two main flavors: (1) long short term memory networks (LSTM) (Hochreiter and Schmidhuber 1997) and (2) gated recurrent units (GRU) (Cho et al. 2014); with the latter being essentially a simplification of the former in order to reduce necessary parameters and thereby training time. Systematic comparisons of LSTMs and GRUs found that none is superior in general, but both showed problem-specific benefits (Zheng et al., 2019, [S8]), Li et al. (2019). In this meta-study, we observe the same

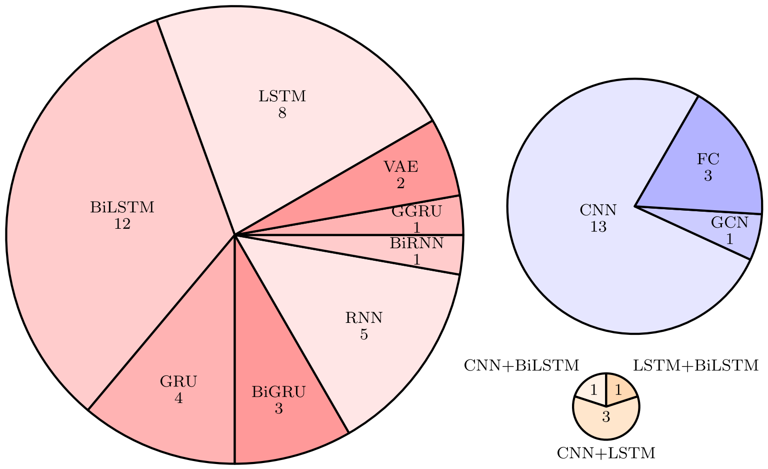

Fig. 8 Proposed model topologies across primary studies grouped into recurrent, feedforward or combined topologies

trend with eight primary studies applying LSTMs in their experiments, and seven proposing GRUs. RNNs face another potential shortcoming due to their sequential processing. They can, within a sequential input, only reason about the input data that they have been exposed to so far and not the remaining part of the input. This problem becomes relevant for encoder-decoder (aka sequence to sequence) problems like translating text from one language into another since output is produced synchronous to the input. The solution is a bidirectional topology consisting of two RNNs (e.g., BiRNN, BiLSTM or BiGRU) that process the sequence simultaneously from both sides and to combine their current hidden states per step upon inference. When performing a classification task on full input sequences, however, typically only the last hidden state, representing the whole sequence, is used for the classification. Vulnerability analysis is such a classification problem and one should not expect substantial benefits from bidirectional topologies. However, 11 primary studies utilize a BiLSTM, 3 use a BiGRU and 1 uses a vanilla BiRNN. Li et al. (2021) ([S7]) found that bidirectional LSTMs (BiLSTM) and GRUs (BiGRU) slightly outperformed their unidirectional equivalents (e.g., LSTM and GRU) reducing error rates by some tenth of percentage points for accuracy, precision and F-measure Zheng et al. (2019) and Li et al. (2021) ([S8], [S7]). In the study of Fang et al. (2019) ([S16]), BiLSTM and LSTM perform similarly having their largest divergence in 1.3 % F-measure in one experiment. One study even found that a vanilla BiRNN outperformed BiLSTM and BiGRU topologies on their dataset (Zheng et al., 2019, [S8]). Another powerful concept proposed to overcome the vanishing gradient problem of long inputs in RNNs is attention. It allows a network to learn which former inputs and their respective hidden states are more and which are less relevant for a given task. Attention has been considered by two primary studies (Zhou et al. 2019 [S23], Liu et al. (2020) [S11]). The authors found that using attention resulted in a 14% higher F-measure.

Deep recurrent topologies The column "Depth" in Table 3 refers to the number of neural network layers employed in a model topology. We counted the main layers of the best performing model topology (marked bold in column "Topology") excluding layers that do not directly contribute to the representation learning ability of a topology, such as embedding, dropout, and classifier layers. Eight primary studies utilize only one main layer, while others propose up to six RNN layers. Li et al. (2018) ([S5]) studied the influence of topology depth in terms of BiLSTM layers. They found that two to three BiLSTM layers yield the highest F-measure and that topologies beyond six layers drastically dropped in terms of F-measure. Deep BiLSTM layers also performed best in terms of highest F-measure in Li et al., (2019, [S12]). While inputs are typically processed sequentially, e.g., layer by layer, in these deep topologies, also skip connections and short circuit branches have been studied (Duan et al. 2019, [S25]).

Feedforward topologies In contrast to recurrent topologies, feed forward topologies are designed for processing all inputs at once. Fully connected neural networks, also known as dense networks or multilayer perceptron (MLP), have connections among all neurons of two successive layers. This fully connected nature makes them ideal classifiers often used in combination with other feedforward or recurrent topologies. However, fully connectedness also goes along with large numbers of parameters making them computationally expensive and memory intensive and therefore hard to scale in depth and width. A specific form of feedforward network is the convolutional neural network (CNN) that uses convolution operations, which aggregate units in the same spatial region and operate with substantially less parameters. CNNs were originally developed for processing n-dimensional arrays (Fukushima et al. 1983) and primary studies frequently propose them in comparison to

RNNs. Hybrid topologies, proposed by five primary studies, combine feedforward layers with recurrent layers. However, Wu et al. (2017, [S28]) found that a CNN yielded higher accuracies than a hybrid topology in their study. The depth, number of hidden layers, and width, e.g., number of neurons per layer, of feedforward topologies vary greatly and are typically chosen in an empirical manner (Abaimov and Bianchi 2019, [S24], Yan et al. 2018 [S29]). In contrast to recurrent topologies, feed forward topologies, including CNNs, expect a static input size. However, samples in the respective datasets typically vary in length and may be longer or shorter than the expected input size of the model. A standard strategy to handle these mismatches is padding shorter inputs with a fix value and truncating longer inputs to the expected input size of the network. This strategy has, e.g., been applied by Wu et al. (2017); Li et al. (2021); Lin et al. (2019). This strategy still leaves the problem of deciding how large the input of the model shall be, e.g., simply choosing the size of the largest training sample does not guarantee that test data will not contain a larger sample, while truncating to samples average length means that potential relevant information will be cut away. Cui et al. (2019) ([S14]) evaluated various input lengths for a CNN and found that models performed better on larger input vectors at the cost of increasing resource consumption for training.

Topologies for graph processing The majority of primary studies transform their intermediate program code graphs into input sequences or matrices. However, five studies employ a topology that can directly operate on graphs as input. In these studies, the graph is processed in the embedding layer of the respective network (cp. Section 5.3). The structure2vec framework (used by two studies Xu et al. 2017; Gao et al. 2018) iteratively creates an embedding to be further processed in a fully connected network, the Graph-GRU uses a specific recurrent layer, and the GCN uses a specific convolutional layer as first processing layer operating on graphical input data. Authors found the Graph-GRU network to be superior when compared to a BiLSTM network with an attention mechanism (Zhou et al. 2019, [S23]). The encoded graph tensor proposed by Duan et al. (2019) ([S25]) uses several convolutional layers for processing. Both networks employ further convolutional and max-pooling layers following the initial embedding to reduce feature dimensionality and several fully connected layers for the final classification task.

Representation learning vs. similarity-based search A majority of 30 primary studies propose an approach that is called representation learning meaning that a model is trained towards identifying and extracting the relevant information from input data and then performing a classification. This classification is realized by an algorithm applied in the last or penultimate layers of the trained model. The most common classifier consists of one or several fully connected neural network layers (FC) (cp. column "Output" of Table 3). Alternative classification approaches are Random tree (RT) and random forest (RF). Random trees try to find the best splitting feature or predictor from a randomly selected subset of features at each node. Random forests use decision trees with a randomly sampled subset of the full dataset.

In contrast to representation learning, there exists the possibility to solely extract and potentially store relevant features of known vulnerability instances and to compare these "signatures" with each new sample that is analyzed. This approach is called similarity-based search since it essentially constitutes a comparison of feature vectors where similar ones are considered a match. Similarity-based search is less computationally intensive than representation learning since only the feature extractor is needed, which can often be reused from a related task. For example, a pre-trained word embedding may be used as a feature extractor for source code. However, the drawback of this approach is that no abstraction of the analyzed concept will be learned. Similarity-based search may still be worthwhile when no sufficient training data or computational resources are available. Two studies propose a similarity-based analysis (Gao et al. (2018) [S1], Xu et al. (2017) [S27]). Gao et al. propose a cross-platform binary vulnerability analysis based on the structure2vec embedding (Gao et al. 2018, [S1]). Their model computes an embedding vector per object code file to be analyzed as well as for each vulnerability to be detected. Vulnerabilities are retrieved from the CVE database (National Vulnerability Database 2020) and their implementation is derived from the Genious-tool by Feng et al. (2016). The actual analysis is then merely a vector comparison, computed as cosine similarity, between all files and all vulnerability embeddings. Dam et al. (2018) ([S3]) propose deep learned features to compare methods of different software projects. A LSTM is trained on code snippets represented as token sequence and output vectors which represent the distribution of semantics of a code token. The vectors of all code tokens are saved in a codebook as so-called global features and can be compared to others by a centroid assignment.

Minimizing generalization error Data augmentation techniques can help to increase the generalization performance and robustness of a trained model, feedforward as well as recurrent, by adding plausible deviations to the training data, e.g., changes to code samples (Cui et al. (2018) [S13], Cui et al. (2019) [S14]), or adding noise (Li et al. 2019, [S12]). An alternative and complementary approach to improve generalization performance and to prevent over-fitting of a model is adding a dropout mechanism (Li et al. (2018) [S5], Fang et al. (2019) [S16]). Models with Dropout randomly disable connections among neurons during training forcing the model to compensate for the missing connections and thereby becoming more robust. The Dropout ratio refers to the proportion of connections to be randomly disabled. Figure 9 shows that 16 primary studies apply dropout with ratios ranging from 20% to 50%, with 50% being to most common choice.

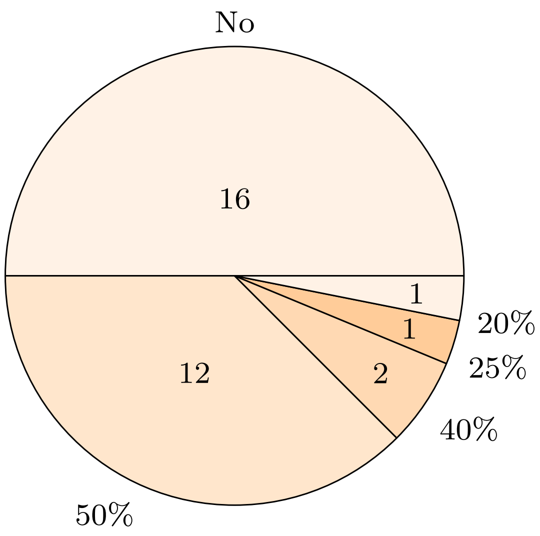

Fig. 9 Extend of dropout within studies using the concept

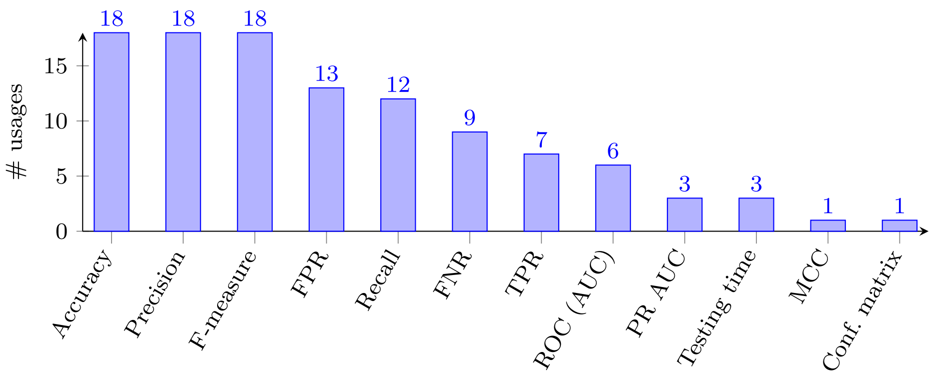

Fig. 10 Used evaluation metrics in the primary studies

Hyper-parameter optimization The adaptation of deep learning methods to a given problem is essential for their performance, but only a quarter of the studies (7 out of 32) perform and describe a systematic hyper-parameters optimization, e.g., a grid search (Cheng et al. 2019[S26]).

Answer to RQ4: The predominant model topology with almost two thirds share are recurrent neural networks used due to the sequential nature of program code and the similarity of source code to natural language text. Graph-based and feedforward topologies are underrepresented and could be further considered in future work.

5.5 RQ5: Evaluation of Proposed Models

This research question explores the evaluation of the proposed approaches. The basis for the evaluation is the availability of ground truth labels (cp. column "Ground truth" of Table 3) that we discuss in Section 5.2 and that serves not only in the supervised training process but also as ground truth for computing evaluation metrics. The last column "Artifacts" of Table 3 provides links to code and datasets of the primary studies if available.

Quality of the evaluation The training and also the evaluation can only be as precise as the utilized labels. There are several difficulties in generating a sufficient and high-quality amount of training and evaluation samples (cp. Section 5.2). For example, labels may not reflect the ground truth due to mistakes in the labeling process. To test the quality of their evaluation, Li et al. (2020) ([S18]) tried to identify vulnerabilities in projects without known ground truth. They labeled 200 program files manually for a qualitative evaluation and used mediocre results by manually checking only false positives as subset. This emphasizes the need to pay great attention to the labeling process and inv esting as a community into high-quality benchmarks.

Metrics We collected the metrics used across all reported experiments of the primary studies (cp. Fig. 10). An often reported metric is accuracy measuring the closeness of a measurement to the true value (International Organization for Standardization 2020). While 18 primary studies report accuracy, it is not a proper evaluation metric for imbalanced datasets, such as many of those evaluated by these studies (cp. Figure 5). In a typical evaluation procedure, a dataset is divided into training, validation, and test splits. That is the test split inherits the imbalance of the overall dataset resulting in a situation where the more dominantly trained classes are also more dominantly tested. This leads to an overly high accuracy neglecting the performance for minority classes, in our case vulnerable code samples. A simple solution is computing an accuracy per class first and to average those accuracies into a so-called class-averaged accuracy. Three primary studies have reported this metric. Other appropriate metrics for evaluating unbalanced datasets are precision, recall and the F-measure, a weighted mean of the former. One or multiple of these measures are reported by 25 primary studies. Precision area under curve (PR AUC) is deviated from the precision recall curve. The receiver operating characteristic (ROC) AUC is the area under the curve of true positive rate as a function of false positive rate. The precision recall curve highlights the skewed data, while the ROC curve concentrates more on the performance (Branco et al. 2016). With ROC, the imbalance is not taken into account. The measure describes the proportion between TPR and FPR. These metrics are ratios of correctly and incorrectly predicted samples and are calculated independently on one side of the confusion matrix meaning that class skew does not influence them (Fawcett 2006). Testing time reported by three studies aims to compare resource usage when applying a previously trained model in production. Matthews correlation coefficient (MCC) is a reliable statistical rate, because all four confusion matrix categories are used equally in its calculation (Chicco and Jurman 2020). MCC is also invariant to class swapping in contrast to the F-measure, which varies when binary classes are accidentally renamed (Chicco and Jurman 2020). This metric is only used once. For a multi-class problem, the confusion matrix between all the vulnerability types is meaningful. For the use case of vulnerability detection, MCC and the confusion matrix are good working measures for binary classification rather than using accuracy and F-measure (Chicco and Jurman 2020). Note that trusting only one measure is often not meaningful, one should always consider several metrics.

Cross-validation For evaluation, more than half of the studies use cross-validation, either as 1-fold, 5-fold or 10-fold cross-validation. Thereby, the number of folds refers to the number of completely repeated training processes, e.g., a 10-fold cross-validation means 10 completely trained models. Twenty primary studies apply such a cross validation strategy, while twelve do not explicitly describe or mention it.

Comparative evaluation Since studies use varying datasets and varying proportions of vulnerable to non-vulnerable examples, we could not compare their raw results to one another. To still be able to contrast studies' approaches, column "Topology" of Table 3 highlights the best performing model topology across a study's experiments in bold. Li et al. (2019)'s study compares a large set of topologies for vulnerability analysis, e.g., FCNN, CNN, LSTM, GRU, BiLSTM, and BiGRU. The comparative study is performed on a dataset of 126 CWE-types represented by 811 security-related C/C++ library function calls (Li et al. 2021, [S7]). The authors found that bidirectional recurrent networks (e.g., BiLSTM and BiGRU) trained on source code accompanied by data and control dependencies (c.p. Section 5.3) resulted in the highest precision and F-measures, but closely followed by feedforward networks (e.g. FCNN and CNN). In total, eight primary studies compare recurrent and feedforward topologies in their experiments. Four of these found recurrent topologies to be superior, while the other identified feedforward topologies as superior for code analysis. Approaches for graph processing use both types of topologies, e.g., Graph-GRUs are based on recurrent GRU layers, while GCNs are based on feedforward CNN layers. We conclude that both topology types are suitable for code analysis.

Answer to RQ5: Various measurements are used for evaluation. Well suited are the MCC, ROC and F-measures. Considering class imbalance is important.

5.6 RQ6: Model Generalizability

Cross-project learning Cross-project learning refers to training a model with program code stemming from one set of software projects, while later analyzing program code from new and unseen projects. To successfully train a generalizing model, typically a substantial amount of representative training data stemming from various projects and many developers is needed. There are manifold levels of generalization in this context that can make a model wider applicable, but also harder to train, e.g., differing application domains or differing development methodologies. Among the primary studies, 18 train and test solely with code samples from one and the same project or generated by the same synthetic sample generator, while seven train on multiple datasets but do not explicitly separate projects for testing. The remaining seven primary studies propose cross-project learning (cp. Fig. 11) in different approaches. While ten primary studies approach cross-project learning solely from a dataset perspective (Russell et al. 2018, [S2]), e.g., training on multiple projects and testing on others, there are also two studies that propose a specific set of "global" features for cross-project learning (Dam et al. (2018) [S3], Zou et al. (2019) [S6]). Global features shall represent some broader semantics about several program slices as high-level view, while local features shall represent individual statements. Lin et al. (2018) ([S30]) compared cross-project learning methods (Li et al. (2019) [S12], Li et al. (2020) [S18], Lin et al. (2018) [S30], Nguyen et al. 2019 [S10], Pechenkin et al. (2018) [S4]). Similar to cross-project learning is cross-version learning proposed by Dam et al. (2018) ([S3]), where the progression over time is the main subject of investigation and new code is introduced in later versions.

Transfer learning Transfer learning refers to a two-stage training procedure originally developed for increasing performance on problems with limited training data. In a first step,

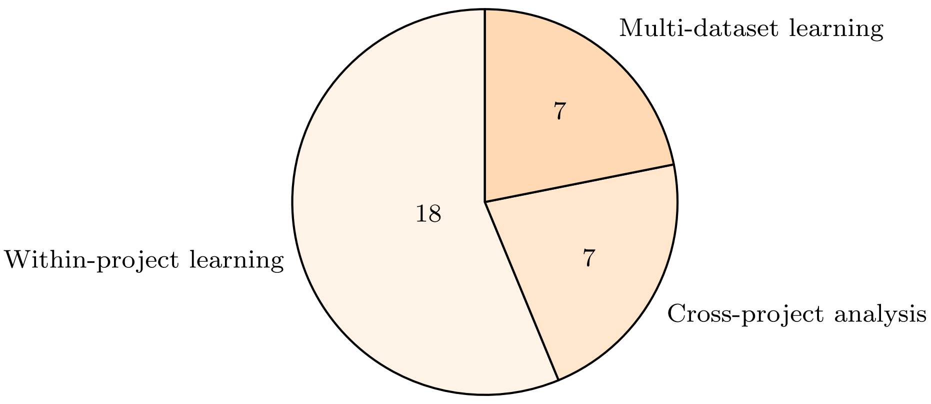

Fig. 11 Consideration of cross-project prediction

a model is pre-trained with a larger, similar dataset to convergence. In a second step, this pre-trained model is taken and then trained with the actual dataset in a procedure called fine-tuning. Thereby, depending on the available training data in the actual dataset, a stronger or softer fine-tuning in terms of retrained parameters and update strength, i.e., learning rate, is chosen. This procedure is very popular for computer vision problems and recently also became popular for natural language processing (Mou et al. 2016). Lin et al. (2018) ([S30]) compare transfer-learning methods and found that representations learned in this manner are more effective than traditional code metrics. In a follow-up study, Lin et al. compared a transfer-learned BiLSTM network with a BiLSTM network trained from scratch showing that the former outperformed the latter in terms of precision and recall.

Answer to RQ6: Cross-project learning is only considered by a quarter of studies, but important when aiming for a real adoption of methods by users. Cross-project learning as well as transfer learning need more investigation in future studies.

6 Trends and Future Directions

Better vulnerability differentiation Proposed methods should evolve from today's predominant binary vulnerable vs. non-vulnerable classification to a vulnerability type classification or ranking (Harer et al. 2018, [S9]). The CWE catalog currently lists 839 individual vulnerability types, while the most elaborate primary study merely aimed to distinguish 40 vulnerabilities. Primary studies acknowledge this shortcoming and plan to adopt their works to more types of vulnerabilities (Xu et al. 2018 [S31], Wu et al. 2017 [S28], Fang et al. (2019) [S16]) or to introduce patterns uncovering multiple types of vulnerabilities (Zou et al. 2019, [S6]). CWE vulnerabilities are partly defined in taxonomic relationship, i.e. more abstract parent CWEs and derived child CWEs. Note that CWEs are based on human-defined categorization and may change from time to time. It may be helpful to exploit these relationships for analysis and presentation of results, e.g., results with limited reliability could be propagated to the parent class.

Object code analysis as fall back On the one hand, vulnerability analysis tools shall intensively be used by software developers to make their source code more secure from the beginning; and source code clearly is the richer information source containing, e.g., identifier names and comments. On the other hand, the ability to analyze binary files enables an analysis of software for which source code is not publicly available. For example, security experts, would be able to study a wide variety of software and users, could establish more trust in their tools. Accordingly, the authors of multiple primary studies argue that in a future extension they aim to expand their work towards object code Liu et al. (2020); Dam et al. (2018); Li et al. (2018); Lin et al. (2019); Liu et al. (2019).

Utilizing unlabeled program code and additional data sources The success and popularity of open-source software gives researchers access to unprecedented amounts of source code written in different languages, targeting different domains and being of varying quality. Future studies should make more use of this valuable resource. Therefore, they have to, however, overcome the problem of vulnerability labeling. Manual labeling whole projects seems unfeasible and using other analysis approaches, such as static code analysis, will limit labeled classes to the capabilities of the respective approach. A potential solution could be unsupervised approaches analyzing program code for uncommon and therefore potentially vulnerable code constructs. Another direction that future studies should explore is utilizing additional datasources upon program code analysis. These can be project-specific datasources, such as version control information, issue tracker information and requirements, in order to identify atypical deviations from specifications and processes. Also general information, such as discussion of programming questions on platforms like Stack Overflow or security bulletins should be utilized.

Towards program code embeddings We found that primary studies use simple bag of words approaches or word embeddings to encode their input data, typically a token stream. These encodings are relatively simple to apply, but have substantial shortcomings. Bag of words approaches neglect semantics among tokens. Word embeddings are able to capture these semantics, but require training with a large representative corpus. When analyzing program code, researchers can decide between (1) training their own embedding meaning that they need to assemble a large corpus of program code or (2) reusing an actionable embedding pretrained on large text corpora like Wikipedia meaning that the embedding is not specifically trained to capture program code semantics. We argue that training specific program code embeddings with large corpora of representative code is an open research question that could substantially improve analysis results. Another shortcoming of word2vec word embeddings is that they can only embed previously trained tokens. While the grammar of a programming language is rather static, user defined identifiers like function and variable names can vary a lot and have an unlimited amount of neologisms (Chen and Monperrus 2019). The FastText word embedding (Bojanowski et al. 2017) overcomes this shortcoming by training not only the actual word but also its character n-grams, i.e., previously unseen words can still be encoded. The latter was not used in any primary study.

Other topologies for analyzing program code We found that primary studies prefer recurrent topologies over feedforward topologies in their experiments, which is motivated by the fact that code is of sequential nature and variable length making recurrent topologies more suitable than standard feedforward topologies including CNNs. However, even the most advanced recurrent topologies suffer from two fundamental problems: (1) recurrence makes their training in large parts sequential and therefore slow; and (2) long input sequences lead to vanishing gradients in the training process. Feedforward Transformer topologies have been proposed to analyze input data of variable length while overcoming these problems (Parmar et al. 2017). Transformers employ sophisticated attention mechanisms and positional encodings to compensate for the missing recurrence and have been demonstrated to substantially outperform RNNs in natural language processing regarding runtime and memory (Botha et al. 2017). We argue that Transformers should also be studied for vulnerability analysis. Furthermore, we found little research on processing program code as graph omitting the typical transformation into input sequences or matrices. A graphical representations seems more natural for representing program code given the underlying control and data flows. We argue that representing and analyzing program code in a graphical way should be a major focus of future research.

Explainability of analysis results Despite its often unprecedented inference quality facilitating more precise analysis results, representation learning with deep neural networks is also criticized as black box, nontransparent and hard to interpret (Selvaraju et al. 2017). Explainability of model decisions is of high importance for vulnerability analysis in order to support developers in understanding and fixing discovered problems (Li et al. (2021) [S7], Wu et al. 2017 [S28], Zhou et al. (2019) [S23], Lin et al. (2019) [S17]). An essential precondition is having fine-grained analysis results, e.g., highlighting problematic code tokens (Russell et al. , [S2]), rather than declaring an entire code sample vulnerable. For example, layer relevance propagation, an explanation technique propagating the prediction of neural networks layer-wise back to its inputs, could be utilized to report which tokens influenced the current decision of a model (Warnecke et al. 2020). Such methods would allow highlighting the most problematic code locations to a user and to guide further inspection and should be explored for vulnerability analysis (Zou et al. (2019) [S6], Li et al. (2019) [S12]). As a future exercise it would also be interesting to deliver an actual explanation of how a given code sample is vulnerable and how an identified weakness may be effectively fixed.

Cross-language and cross-architecture learning Cross-language learning is the abstraction of vulnerability patterns from a specific language in order to facilitate language-agnostic vulnerability analysis of source code Zou et al. (2019) [S6], Dam et al. (2018) [S3]). None of the primary studies approach vulnerability analysis in this manner. However, it seems a relevant future research topic as it promises more versatile and universal models, and could uncover a more holistic view on vulnerability types, e.g., which are language specific and which are language agnostic. A related concept is cross-architecture learning aiming to abstract object-code from the architecture it was compiled for and producing models that are applicable to a wide variety of object code (Gao et al. (2018) [S1], Xu et al. (2017) [S27]).

Towards usable analysis systems Methods should evolve from proprietary implementations on tailored datasets to end-to-end software analysis tools applicable in practice and supporting software developers in the development of secure software. There is still room for improvement of DL-based methods as well as SAST tools (Johnson et al. 2013; Christakis and Bird 2016). An end-to-end software analysis tool shall be able to analyze projects created in a large variety of programming languages and with differing development methodologies. This includes delivering pre-trained models that generalize from a specific software project and potentially also from an application domain. For the training of such models a substantial and representative training corpus is needed.

7 Threats to Validity

Below, we discuss threats to the validity of our meta study grouped according to four commonly used categories: construct, internal, external, conclusion validity (Easterbrook et al. 2008).

The **construct validity** threats concern the relationship between the theory and the application. We carefully defined our search terms and selection criteria according to Kitchenham and Charters's guidelines, but during construction of these we may have missed an important keyword. We tried to mitigate this threat (i) by performing a search on Google Scholar as meta search engine with richer and broader results; and (ii) by conducting a citation check forward and backward on the primary studies identified up to this point. Furthermore, we limited our meta-study to English-language publications as English is the common language for publishing in computer science. Moreover, this field is fast-evolving, so the number of relevant studies can grow fast. At the time of completing this meta-study (May 2020), we consider our survey complete.

The main threats to **internal validity** are author bias, author influence and the understandability of the written reports leading to a possible inaccuracy in data analysis and extraction. To overcome the mentioned issues, we documented all executed steps in a protocol and re-checked these steps between the authors. Our search strategy included a filter on publication title and abstract in an early phase of the study search process. We used a predefined search string ensuring that we only searched for primary studies having their main focus on vulnerability analysis in the deep learning field. Therefore, studies that propose a more general deep learning on code classification approach with different classes than vulnerability types may have been excluded by this filter, but could be easily adapted to this application. This is valid, because further work is needed for their adaption.

External validity refers to the representativeness of results. To overcome potential threats in this direction, we used different representative publication repositories. Given the fact that we study a relatively new field of research and there is not an enormous amount of relevant papers, we restricted quality analysis of primary studies to a minimum. That means that some primary studies are short papers and important details of their approaches are not reported making it hard to fully compare them against others. Therefore, we had to infer certain information during the data extraction process, but intensively discussed these cases among the authors of this survey. Our discussion is not entirely complete due to this missing information. We would have liked to collect additional information on utilized models, e.g., the number of tokens per input sample or the number of parameters per layer across topologies.

Finally, **conclusion validity** involves the threat of deriving correct conclusions from this literature review. We counteract to this by a discussion between the authors to reach consensus for each conclusion. However, the reader could come to her own conclusion.

8 Conclusions

In this meta-study, we surveyed publications on deep learning assisted vulnerability analysis of source code and object code. We present the evolution from traditional machine learning to deep learning approaches for this application that took place in the last three to four years. Training data forms the basis for the studies and was inspected in Section 5.2. We also provide different categorization schemata according to the studies' input pre-processing and the level of granularity in Section 5.3 to show the differences across the primary studies. We compared studies regarding code representation, topology and evaluation corresponding to the dataset structure. This work also advises which topology and evaluation metrics are suitable for follow-up studies. Furthermore, we point out limitations of primary studies and discuss future research directions, such as more extensive dataset labeling, cross-project prediction, processing program code as graph, and providing explainable code analysis results.

Appendix A: Datasets for Vulnerability Analysis

Table 4 Overview of utilized datasets

No.	Dataset	Description	Samples				Labels		Format	URL	Published	Metric
			language	origin	#	granularity	origin	granularity				
1	SARD (Black 2018)	collection of code samples, originally developed for evaluating static analysis tools, continuously updated by NIST*^{2}	7 diff. lang.	syn. & real	177,184	file	911 CWEs	statement	raw	link	2006	–*^{1}
2	Static Analysis Test Suite (Chatzieleftheriou and Katsaros 2011)	developed as static analysis benchmark dataset, also applicable for DL methods	C	real	871	method	30 CWEs	statement	raw	Link	2011	F 85,0
3	PHP Security Vulnerability Dataset (Walden et al. 2014)	manually curated, raw and replicated version with additional code metrics	PHP	real	2,942	file	6 CWEs	token string	Rda, arff	link	2014	Rec. 78,1
4	Function representation learning (Lin et al. 2017)	code snippets collected from 3 FOSS projects: FFmpeg, LibTIFF, LibPNG, includes additional code metrics	C/C++	real	7,202	file	329 CVEs	file	raw	link	2017	Rec. 47,0
5	Juliet Test Suite for C/C++ (Boland and Black 2012)	selection of examples from dataset No. 1, latest version 1.3, created by NIST *^{2}	C/C++	syn.	64,099	method	118 CWEs	method	raw	link	2017	–*^{1}
6	Code Gadget Database (CGD) (Li et al. 2018)	only CWE-119 and CWE-399, labeled data from SARD and CVE database *^{2}	C/C++	syn. & real	61,638	method	2 CWEs	method	raw	link	2017	F 88,2
7	Juliet Test Suite for Java (Boland and Black 2012)	selection of examples from dataset No. 1, latest version 1.3, created by NIST *^{2}	Java	syn.	28,881	method	112 CWEs	method	raw	link	2017	–*^{1}
8	SySeVR (Li et al. 2021)	data from 4 FOSS projects: FFmpeg, LibPNG, LibTIFF, VLC, focuses on insecure C/C++ library method calls	C/C++	real	17,893	method	126 CWEs	method	raw	link	2018	F 85,9
9	VulinOSS (Gkortzis et al. 2018)	data from 153 different FOSS projects with the highest number of CVEs, includes additional code metrics	219 diff. lang.	real	38,650	project	17,738 CVEs	project	SQL	link	2018	–*^{1}
10	Transfer Representation Learning (Lin et al. 2018)	data from 6 FOSS projects: LibTIFF, LibPNG, FFmpeg, Pidgin, VLC, Asterik	C	real	40,677	method	609 CVEs	method	raw	link	2018	Pre. 80,0

Table 4 (continued)

No.	Dataset	Description	Samples				Labels		Format	URL	Published	Metric
			language	origin	#	granularity	origin	granularity				
11	Draper VDISC Dataset (Russell et al. 2018)	collected examples from Juliet Test Suite (dataset No. 5) and various FOSS projects	C/C++	syn. & real	1,274,366	method	5 CWEs	method	HDF5	link	2018	F 69,5
12	Manually curated dataset of fixes to vuln. of OSS (Ponta et al. 2019)	selected projects related to SAP products, contains supporting scripts as python notebook	Java	real	1,722	project	binary detection	multi-statement	link	link	2019	–*1
13	Representation learning from multi. domain (Lin et al., 2019)	data from 6 FOSS projects: Asterisk, FFmpeg, LibPNG, LibTIFF, Pidgin, VLC	C	real	39,958	method	binary detection	method	raw	link	2019	Rec. 82,0
14	Source code analysis dataset (Gelman et al. 2019)	dataset contains raw static analysis outputs by tool infer (Facebook 2020)	C/C++	real	3,170	file	raw SA outputs	not pre-processed	raw	link	2019	–*1
15	μVulDeePecker Multiclass Vuln. Dataset (MVD) (Zou et al., 2019)	data is based on SARD and NVD, contains pre-processed program slices as well as the raw source code	C/C++	syn. & real	138,522	file	40 CWEs	multi-statement	raw	link	2019	F 96,28
16	Devign (Zhou et al., 2019)	manually labeled examples with data from 2 FOSS projects: qemu and FFmpeg	C/C++	real	25,872	multi-statement	binary detection	multi-statement	csv, json	link	2019	F 73,26
17	LAVDNN (Li et al., 2019)	contains method names for classification, with additional noise data	Python, C/C++	real	8,027	statement	binary detection	statement	txt	link	2019	F 89,6
18	XSS Payload list (Abaimov and Bianchi, 2019)	list with vulnerable SQL statements for detecting SQL injections	SQL	real	6,586	statement	1 CWE	statement	txt	link	2019	Rec. 96,2
19	MDSeqVAE (Le et al., 2019)	pre-processed examples of dataset No. 6 with doc2vec embedding	object code	syn. & real	32,281	multi-statement	2 CWEs	method	txt	link	2019	F 87,1
20	VulDeeLocator (Li et al., 2021)	contains code fragments with syntactic and semantic information created by the LLVM tool (Lattner and Adve 2004)	C/C++	syn. & real	119,782	method	558 CVEs and 11 CWEs	line of code	raw	link	2020	F 97,0

[1]No benchmark metric available, because the dataset has been published without analysis results. We only considered the primary publication. – *2 (National Vulnerability Database 2020)

Acknowledgements This work was supported by Deutsche Forschungsgemeinschaft (DFG; Grant No: MA 5030/3– 1) and the German Federal Ministry of Education an Research (BMBF; Grant No: 01IS18074E).

Funding Open Access funding enabled and organized by Projekt DEAL.

Further Reading

[S1] Gao J, Yang X, Fu Y, Jiang Y, Shi H, Sun J (2018) VulSeeker-pro: Enhanced Semantic Learning Based Binary Vulnerability Seeker with Emulation. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2018, pp 803–808, DOI 10.1145/3236024.3275524, URL http://doi.acm.org/10.1145/3236024. 3275524, event-place: Lake Buena Vista, FL, USA

[S2] Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M (2018) Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), pp 757–762

[S3] Dam HK, Tran T, Pham TTM, Ng SW, Grundy J, Ghose A (2018) Automatic feature learning for predicting vulnerable software components. IEEE Transactions on Software Engineering pp 1–1, DOI 10.1109/TSE.2018. 2881961

[S4] Pechenkin A, Demidov R (2018) Applying Deep Learning and Vector Representation for Software Vulnerabilities Detection. In: Proceedings of the 11th International Conference on Security of Information and Networks, ACM, New York, NY, USA, SIN '18, pp 13:1–13:6, DOI 10.1145/3264437. 3264489, URL http://doi.acm.org/10.1145/3264437.3264489, eventplace: Cardiff, United Kingdom

[S5] Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018) VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. Proceedings 2018 Network and Distributed System Security Symposium DOI 10.14722/ndss.2018.23158, URL https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_03A-2_Li_paper.pdf

[S6] Zou D, Wang S, Xu S, Li Z, Jin H (2019) *mu*VulDeePecker: A Deep Learning- Based System for Multiclass Vulnerability Detection. IEEE Transactions on Dependable and Secure Computing pp 1–1, DOI 10.1109/TDSC.2019. 2942930

[S7] Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z (2021) Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing pp 1–1, DOI 10.1109/TDSC.2021. 3051525

[S8] Zheng J, Pang J, Zhang X, Zhou X, Li M, Wang J (2019) Recurrent neural network based binary code vulnerability detection. In: Proceedings of the 2019 2nd International Conference on Algorithms, Computing and ArtificialIntelligence, Association for Computing Machinery, New York, NY, USA, ACAI 2019, p 160–165, DOI 10.1145/3377713.3377738, URL https://doi.org/10.1145/3377713.3377738

[S9] Harer JA, Kim LY, Russell RL, Ozdemir O, Kosta LR, Rangamani A, Hamilton LH, Centeno GI, Key JR, Ellingwood PM, Antelman E, Mackay A, McConley MW, Opper JM, Chin P, Lazovich T (2018) Automated software vulnerability detection with machine learning. arXiv:180304497 [cs, stat] URL http://arxiv.org/abs/1803.04497, arXiv: 1803.04497

[S10] Nguyen V, Le T, Le T, Nguyen K, DeVel O, Montague P, Qu L, Phung D (2019) Deep Domain Adaptation for Vulnerable Code Function Identification. In: 2019 International Joint Conference on Neural Networks (IJCNN), pp 1–8, DOI 10.1109/IJCNN.2019.8851923, iSSN: 2161-4393

[S11] Liu S, Dibaei M, Tai Y, Chen C, Zhang J, Xiang Y (2020) Cyber vulnerability intelligence for internet of things binary. IEEE Transactions on Industrial Informatics 16(3):2154–2163, DOI 10.1109/TII.2019.2942800

[S12] Li R, Feng C, Zhang X, Tang C (2019) A lightweight assisted vulnerability discovery method using deep neural networks. IEEE Access 7:80079–80092, DOI 10.1109/ACCESS.2019.2923227

[S13] Cui Z, Xue F, Cai X, Cao Y, Wang G, Chen J (2018) Detection of Malicious Code Variants Based on Deep Learning. IEEE Transactions on Industrial Informatics 14(7):3187–3196, DOI 10.1109/TII.2018.2822680

[S14] Cui Z, Du L, Wang P, Cai X, Zhang W (2019) Malicious code detection based on CNNs and multi-objective algorithm. Journal of Parallel and Distributed Computing 129:50–58, DOI 10.1016/j.jpdc. 2019.03.010, URL http://www.sciencedirect.com/science/article/ pii/S0743731518308529

[S15] Liu S, Lin G, Han QL, Wen S, Zhang J, Xiang Y (2019) DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection. IEEE Transactions on Fuzzy Systems pp 1–1, DOI 10.1109/TFUZZ.2019.2958558

[S16] Fang Y, Han S, Huang C, Wu R (2019) Tap: A static analysis model for php vulnerabilities based on token and deep learning technology. PloS one 14(11)

[S17] Lin G, Zhang J, Luo W, Pan L, De Vel O, Montague P, Xiang Y (2019) Software Vulnerability Discovery via Learning Multi-domain Knowledge Bases. IEEE Transactions on Dependable and Secure Computing pp 1–1, DOI10.1109/TDSC.2019.2954088

[S18] Li Z, Zou D, Xu S, Chen Z, Zhu Y, Jin H (2021) Vuldeelocator: A deep learning-based fine-grained vulnerability detector. IEEE Transactions on Dependable and Secure Computing pp 1–1, DOI 10.1109/TDSC.2021. 3076142

[S19] Li R, Zhang C, Feng C, Zhang X, Tang C (2019) Locating vulnerability in binaries using deep neural networks. IEEE Access 7:134660–134676, DOI 10.1109/ACCESS.2019.2942043

[S20] Demidov R, Pechenkin A (2018) Vector representation of machine instructions for vulnerability assessment of digital infrastructure components. In: 2018 IEEE Industrial Cyber-Physical Systems (ICPS), pp 835–840, DOI10.1109/ICPHYS.2018.8390815

[S21] Niu W, Zhang X, Du X, Zhao L, Cao R, Guizani M (2020) A deep learning based static taint analysis approach for IoT software vulnerability location. Measurement 152:107139, DOI 10.1016/j.measurement. 2019.107139, URL http://www.sciencedirect.com/science/article/pii/S026322411931005X

[S22] Xiaomeng W, Tao Z, Runpu W, Wei X, Changyu H (2018) CPGVA: Code Property Graph based Vulnerability Analysis by Deep Learning. In: 2018 10th International Conference on Advanced Infocomm Technology (ICAIT), pp 184–188, DOI 10.1109/ICAIT.2018.8686548

[S23] Zhou Y, Liu S, Siow J, Du X, Liu Y (2019) Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. NeurIPS2019 p 11

[S24] Abaimov S, Bianchi G (2019) Coddle: Code-Injection Detection With Deep Learning. Ieee Access 7:128617–128627, Doi 10.1109/Access.2019. 2939870

[S25] Duan X, Wu J, Ji S, Rui Z, Luo T, Yang M, Wu Y (2019) Vulsniper: focus your attention to shoot fine-grained vulnerabilities. In: Proceedings of the 28th International Joint Conference on Artificial Intelligence, AAAI Press, pp 4665–4671

[S26] Cheng X, Wang H, Hua J, Zhang M, Xu G, Yi L, Sui Y (2019) Static detection of control- ow-related vulnerabilities using graph embedding. In: 2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS), pp 41–50, DOI 10.1109/ICECCS.2019.00012

[S27] Xu X, Liu C, Feng Q, Yin H, Song L, Song D (2017) Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, New York, NY, USA, CCS '17, pp 363–376, DOI 10.1145/3133956.3134018, URL http://doi.acm.org/10.1145/ 3133956.3134018, event-place: Dallas, Texas, USA

[S28] Wu F, Wang J, Liu J, Wang W (2017) Vulnerability detection with deep learning. In: 2017 3rd IEEE International Conference on Computer and Communications (ICCC), pp 1298–1302, DOI 10.1109/CompComm.2017. 8322752

[S29] Yan R, Xiao X, Hu G, Peng S, Jiang Y (2018) New deep learning method to detect code injection attacks on hybrid applications. Journal of Systems and Software 137:67–77, DOI 10.1016/j.jss.2017.11.001, URL https: //linkinghub.elsevier.com/retrieve/pii/S0164121217302571

[S30] Lin G, Zhang J, Luo W, Pan L, Xiang Y, Vel OD, Montague P (2018) Cross-Project Transfer Representation Learning for Vulnerable Function Discovery. IEEE Transactions on Industrial Informatics 14(7):3289–3297, DOI 10.1109/TII.2018.2821768

[S31] Xu A, Dai T, Chen H, Ming Z, Li W (2018) Vulnerability Detection for Source Code Using Contextual LSTM. In: 2018 5th International Conference on Systems and Informatics (ICSAI), pp 1225–1230, DOI 10.1109/ICSAI. 2018.8599360

[S32] Le T, Nguyen T, Le T, Phung D, Montague P, Vel OD, Qu L (2019) Maximal divergence sequential autoencoder for binary software vulnerability detection. In: International Conference on Learning Representations, URL https://openreview.net/forum?id=ByloIiCqYQ

References

National Institute of Standards and Technology (2020) Vulnerability - Glossary | CSRC. https://csrc.nist.gov/glossary/term/vulnerability

National Vulnerability Database (2020) NVD - National Vulnerability Database - Search and Statistics. https://nvd.nist.gov/vuln/search

Kumar C, Yadav DK (2017) Software defects estimation using metrics of early phases of software development life cycle, vol 8, pp 2109–2117. https://doi.org/10.1007/s13198-014-0326-2. https://ideas.repec.org/a/spr/ijsaem/v8y2017i4d10.1007_s13198-014-0326-2.html

Allamanis M, Barr ET, Devanbu P, Sutton C (2018) A survey of machine learning for big code and naturalness. ACM Comput Surv 51(4):81:1–81:37. https://doi.org/10.1145/3212695

The MITRE Corporation (2020) CWE - CWE List Version 4.0. https://cwe.mitre.org/data/index.html

Allamanis M, Barr ET, Devanbu P, Sutton C (2018) A survey of machine learning for big code and naturalness. ACM Comput Surv (CSUR) 51(4):81

Ucci D, Aniello L, Baldoni R (2019) Survey of machine learning techniques for malware analysis, vol 81, pp 123–147. DOI:10.1016/j.cose.2018.11.001. https://www.sciencedirect.com/science/article/pii/S0167404818303808

Lin G, Wen S, Han QL, Zhang J, Xiang Y (2020) Software vulnerability detection using deep neural networks: a survey. Proc IEEE 108(10):1825–1848. https://doi.org/10.1109/JPROC.2020.2993293

Choi YH, Liu P, Shang Z, Wang H, Wang Z, Zhang L, Zhou J, Zou Q (2020) Using deep learning to solve computer security challenges: a survey. Cybersecurity 3(15). https://doi.org/10.1186/s42400-020-00055-5

Berman DS, Buczak AL, Chavis JS, Corbett CL (2019). A survey of deep learning methods for cyber security. Information 10(4). https://doi.org/10.3390/info10040122. https://www.mdpi.com/2078-2489/10/4/122

Guan Z, Bian L, Shang T, Liu J (2018) When machine learning meets security issues: A survey. In: 2018 IEEE international conference on intelligence and safety for robotics (ISR). pp 158–165. https://doi.org/10.1109/IISR.2018.8535799

Ferrag MA, Maglaras L, Moschoyiannis S, Janicke H (2020) Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study, vol 50, p 102419. https://doi.org/10.1016/j.jisa.2019.102419. https://www.sciencedirect.com/science/article/pii/S2214212619305046

Ghaffarian SM, Shahriari HR (2017) Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey. ACM Comput Surv 50(4). https://doi.org/10.1145/3092566

Jie G, Xiao-Hui K, Qiang L (2016) Survey on software vulnerability analysis method based on machine learning. In: 2016 IEEE first international conference on data science in cyberspace (DSC), pp 642–647. https://doi.org/10.1109/DSC.2016.33

Le Q, Boydell O, Mac NameeB, Scanlon M (2018) Deep learning at the shallow end: Malware classification for non-domain experts, vol 26, pp S118–S126. https://doi.org/10.1016/j.diin.2018.04.024. http://www.sciencedirect.com/science/article/pii/S1742287618302032

Cakir B, Dogdu E (2018) Malware classification using deep learning methods. In: Proceedings of the ACMSE 2018 conference, ACMSE '18. Association for Computing Machinery, Richmond, pp 1–5. https://doi.org/10.1145/3190645.3190692

Alon U, Zilberstein M, Levy O, Yahav E (2019) code2vec: learning distributed representations of code. Proc ACM Program Lang 3(POPL):40:1–40:29. https://doi.org/10.1145/3290353

Hellendoorn VJ, Bird C, Barr ET, Allamanis M (2018) Deep learning type inference. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2018. Association for Computing Machinery, Lake Buena Vista, pp 152–162. https://doi.org/10.1145/3236024.3236051

Cambronero J, Li H, Kim S, Sen K, Chandra S (2019) When deep learning met code search. In: Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2019. Association for Computing Machinery, Tallinn, pp 964-974. https://doi.org/10.1145/3338906.3340458

LeCun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521(7553):436–444. https://doi.org/10.1038/nature14539. https://www.nature.com/articles/nature14539

Shin Y, Williams L (2013) Can traditional fault prediction models be used for vulnerability prediction? Empir Softw Eng 18(1):25–59. 10.1007/s10664-011-9190-8

Pradel M, Chandra S (2021) Neural software analysis. Commun ACM. arXiv:2011.07986v2

Hindle A, Barr ET, Su Z, Gabel M, Devanbu P (2012) On the naturalness of software. In: Proceedings of the 34th international conference on software engineering, ICSE '12. IEEE Press, Zurich, pp 837–847

Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs? In: Proceedings of the 2013 international conference on software engineering, iCSE '13. IEEE Press, San Francisco, pp 672–681

Christakis M, Bird C (2016) What developers want and need from program analysis: an empirical study. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, ASE 2016. Association for Computing Machinery, New York, pp 332-343

Lopes CV, Maj P, Martins P, Saini V, Yang D, Zitny J, Sajnani H, Vitek J (2017) Déjàvu: a map of code duplicates on GitHub. Proc ACM Prog Lang 1(OOPSLA):84:1–84:28. https://doi.org/10.1145/3133908

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining GitHub. In: Proceedings of the 11th working conference on mining software repositories, MSR 2014. Association for Computing Machinery, Hyderabad, pp 92–101. https://doi.org/10.1145/2597073.2597074

Komer B, Bergstra J, Eliasmith C (2014) Hyperopt-Sklearn: automatic hyperparameter configuration for scikit-learn. In: Proceedings of the 2014 Python in science conferences, Austin, Texas, pp 32–37. https://doi.org/10.25080/Majora-14bd3278-006. https://conference.scipy.org/proceedings/scipy2014/komer.html

Kitchenham B, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering. Tech. rep. https://www.elsevier.com/__data/promis_misc/525444systematicreviewsguide.pdf

Markovtsev V, Long W (2018) Public git archive: a big code dataset for all. In: Proceedings of the 15th international conference on mining software repositories,MSR '18. Association for Computing Machinery, Gothenburg, pp 34–37. https://doi.org/10.1145/3196398.3196464

Black PE (2018) A software assurance reference dataset: Thousands of programs with known bugs. J Res Natl Inst Standards Technol 123(123005):3

Michaud F, Painchaud F (2008) Language insecurity open source business resource place. Ottawa Publisher, Ottawa. Talent First Network

National Vulnerability Database (2014) NVD - CVE-2014-1912. https://nvd.nist.gov/vuln/detail/CVE-2014-1912

Chen Z, Monperrus M (2019)

Mikolov T, Sutskever I, Chen K, Corrado G, Dean J (2013) Distributed representations of words and phrases and their compositionality. In: Proceedings of the 26th international conference on neural information processing systems, NIPS'13, vol 2. Curran Associates Inc., Lake Tahoe, pp 3111–3119

Pennington J, Socher R, Manning C (2014) GloVe: Global vectors for word representation. In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). Association for Computational Linguistics, Doha, pp 1532–1543. https://doi.org/10.3115/v1/D14-1162. https://www.aclweb.org/anthology/D14-1162

Bojanowski P, Grave E, Joulin A, Mikolov T (2017) Enriching word vectors with subword information. arXiv:1607.04606v2

Kipf TN, Welling M (2017) Semi-supervised classification with graph convolutional networks. In: Proceedings of the 5th international conference on learning representations, Palais des Congr{\'e}s Neptune, Toulon, France, p 14. https://openreview.net/forum?id=SJU4ayYgl

Dai H, Dai B, Song L (2016) Discriminative embeddings of latent variable models for structured data. In: Proceedings of the 33rd international conference on international conference on machine learning, ICML'16, vol 4. JMLR.org, New York, pp 2702–2711

Hopfield JJ (1982) Neural networks and physical systems with emergent collective computational abilities, vol 79, pp 2554–2558. https://doi.org/10.1073/pnas.79.8.2554. https://www.pnas.org/content/79/8/2554. publisher: National Academy of Sciences Section: Research Article

Hochreiter S, Schmidhuber J (1997) Long short-term memory, vol 9, pp 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735. publisher: MIT Press

Cho K, van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2014) Learning phrase representations using RNN encoder–decoder for statistical machine translation. In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). Association for Computational Linguistics, Doha, pp 1724–1734. https://doi.org/10.3115/v1/D14-1179. https://www.aclweb.org/anthology/D14-1179

Li Z, Zou D, Tang J, Zhang Z, Sun M, Jin H (2019) A comparative study of deep learning-based vulnerability detection system, vol 7, pp 103184–103197. https://doi.org/10.1109/ACCESS.2019.2930578

Fukushima K, Miyake S, Ito T (1983) Neocognitron: A neural network model for a mechanism of visual pattern recognition. IEEE Trans Syst Man Cybern SMC 13(5):826–834. https://doi.org/10.1109/TSMC.1983.6313076

Feng Q, Zhou R, Xu C, Cheng Y, Testa B, Yin H (2016) Scalable graph-based bug search for firmware images. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, CCS '16. ACM, New York, pp 480–491. https://doi.org/10.1145/2976749.2978370, event-place: Vienna, Austria

International Organization for Standardization (2020) ISO 5725-1:1994(en), Accuracy (trueness and precision) of measurement methods and results — Part 1: General principles and definitions. https://www.iso.org/obp/ui/#iso:std:iso:5725:-1:ed-1:v1:en

Branco P, Torgo L, Ribeiro RP (2016) A survey of predictive modeling on imbalanced domains. https://doi.org/10.1145/2907070

Fawcett T (2006) An introduction to ROC analysis, vol 27, pp 861–874. https://doi.org/10.1016/j.patrec.2005.10.010. https://linkinghub.elsevier.com/retrieve/pii/S016786550500303X

Chicco D, Jurman G (2020) The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. BMC Genom 21(1):6. https://doi.org/10.1186/s12864-019-6413-7

Mou L, Meng Z, Yan R, Li G, Xu Y, Zhang L, Jin Z (2016) How transferable are neural networks in nlp applications? In: Proceedings of the 2016 conference on empirical methods in natural language processing. Association for Computational Linguistics, Austin, pp 479–489. https://doi.org/10.18653/v1/D16-1046. https://www.aclweb.org/anthology/D16-1046

Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems. pp 5998–6008

Botha JA, Pitler E, Ma J, Bakalov A, Salcianu A, Weiss D, McDonald R, Petrov S (2017) Natural language processing with small feed-forward networks. arXiv:170800214

Selvaraju RR, Cogswell M, Das A, Vedantam R, Parikh D, Batra D (2017) Grad-CAM: visual explanations from deep networks via gradient-based localization. In: 2017 IEEE international conference on computer vision (ICCV), pp 618–626. https://doi.org/10.1109/ICCV.2017.74

Warnecke A, Arp D, Wressnegger C, Rieck K (2020) Evaluating explanation methods for deep learning in security. arXiv:1906.02108 [cs, stat]

Easterbrook S, Singer J, Storey MA, Damian D (2008) Selecting empirical methods for software engineering research. In: Shull F, Singer J, Sjøberg DIK (eds) Guide to advanced empirical software engineering. Springer, London, pp 285–311. https://doi.org/10.1007/978-1-84800-044-5_11

Chatzieleftheriou G, Katsaros P (2011) Test-driving static analysis tools in search of C code vulnerabilities. In: 2011 IEEE 35th annual computer software and applications conference workshops. pp 96–103. https://doi.org/10.1109/COMPSACW.2011.26

Walden J, Stuckman J, Scandariato R (2014) Predicting vulnerable components: software metrics vs text mining. In: 2014 IEEE 25th international symposium on software reliability engineering. pp 23–33. https://doi.org/10.1109/ISSRE.2014.32

Lin G, Zhang J, Luo W, Pan L, Xiang Y (2017) POSTER: vulnerability discovery with function representation learning from unlabeled projects. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security - CCS '17. ACM Press, Dallas, pp 2539–2541. https://doi.org/10.1145/3133956.3138840. http://dl.acm.org/citation.cfm?doid=3133956.3138840

Boland T, Black PE (2012) Juliet 1.1 C/C++ and Java test suite. Computer 45(10):88–90. https://doi.org/10.1109/MC.2012.345. http://ieeexplore.ieee.org/document/6329885/

Gkortzis A, Mitropoulos D, Spinellis D (2018) VulinOSS: a dataset of security vulnerabilities in open-source systems. In: Proceedings of the 15th international conference on mining software repositories, MSR '18. ACM, New York, pp 18–21. https://doi.org/10.1145/3196398.3196454. event-place: Gothenburg, Sweden

Ponta SE, Plate H, Sabetta A, Bezzi M, Dangremont C (2019) A manually-curated dataset of fixes to vulnerabilities of open-source software. In: Proceedings of the 16th international conference on mining software repositories, MSR '19. IEEE Press, Montreal, pp 383–387. https://doi.org/10.1109/MSR.2019.00064

Gelman B, Obayomi B, Moore J, Slater D (2019) Source code analysis dataset. Data Brief 27:104712. https://doi.org/10.1016/j.dib.2019.104712. http://www.sciencedirect.com/science/article/pii/S2352340919310674

Facebook (2020) Infer static analyzer. https://fbinfer.com/

Lattner C, Adve V (2004) Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on code generation and optimization: feedback-directed and runtime optimization, CGO '04. IEEE Computer Society, p 75

Lin G, Zhang J, Luo W, Pan L, Xiang Y, Vel OD, Montague P (2018) Cross- project transfer representation learning for vulnerable function discovery. IEEE Trans Ind Inform 14(7):3289–3297. https://doi.org/10.1109/TII.2018.2821768

Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018) VulDeePecker: a deep learning-based system for vulnerability detection. In: Proceedings 2018 network and distributed system security symposium. https://doi.org/10.14722/ndss.2018.23158, https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss201803A-2Lipaper.pdf

Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z (2021) Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Trans Depend Secur Comput :1–1. https://doi.org/10.1109/TDSC.2021.3051525

Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M (2018) Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE international conference on machine learning and applications (ICMLA). pp 757–762