Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SPECIAL ISSUE

An Empirical Evaluation of Deep Learning-Based Source Code Vulnerability Detection: Representation vs. Models

Abubakar Omari Abdallah Semasaba[1] | Wei Zheng*[1] | Xiaoxue Wu[2] | Samuel Akwasi Agyemang[1] | Tao Liu[3] | Yuan Ge[4]

[1]School of Software, Northwestern Polytechnical University, Xi'an, China
[2]School of Information Engineering, Yangzhou University, Yangzhou, China
[3]School of Computer and Information, Anhui Polytechnic University, Anhui, China
[4]School of Electrical Engineering, Anhui Polytechnic University, Anhui, China

Correspondence
*Wei Zheng, School of Software, Northwestern Polytechnical University Xi'an, China. Email: wzheng@nwpu.edu.cn

Abstract

Vulnerabilities in the source code of the software are critical issues in the realm of software engineering. Coping with vulnerabilities in software source code is becoming more challenging due to several aspects such as complexity and volume. Deep learning has gained popularity throughout the years as a means of addressing such issues. This paper proposes an evaluation of vulnerability detection performance on source code representations and evaluates how Machine Learning (ML) strategies can improve them. The structure of our experiment consists of 3 Deep Neural Networks (DNNs) in conjunction with five different source code representations; Abstract Syntax Trees (ASTs), Code Gadgets (CGs), Semantics-based Vulnerability Candidates (SeVCs), Lexed Code Representations (LCRs), and Composite Code Representations (CCRs). Experimental results show that employing different ML strategies in conjunction with the base model structure influences the performance results to a varying degree. However, ML-based techniques suffer from poor performance on class imbalance handling and dimensionality reduction when used in conjunction with source code representations.

KEYWORDS:
security, software vulnerability detection, deep learning

1 | INTRODUCTION

Software vulnerabilities have an enormous impact and result in security-related risks. These risks cause both reputational and economic damages to companies and people using such software [1]. Due to such circumstances, the early detection of such vulnerabilities is critical. A vulnerability is a weakness in a system, mainly in security system procedures, implementations, or internal controls that can be exploited by external threat sources [2]. A bug is a defect in the system that could lead to a vulnerability [3]. Thus, vulnerabilities are software bugs that are exploitable for malicious intentions [4,5]. Identifying vulnerabilities is quite different from that of defects due to the difficulty of realizing them during normal system operations by users or developers.

In contrast, defects are more accessible to notice [5]. With several software developments and their role in several different aspects of the world, the potential for security issues in software is becoming an emerging challenge. Hackers with a high degree of skill can exploit vulnerabilities found in software to do harmful things such as stealing private information. Using vulnerable between devices is widespread, the risk of security issues is ever-present. This environment has led to a large number of security issues that have affected many users. According to the statistics of the Common Vulnerabilities and Exposures (CVE) organization [6], the number of discovered vulnerabilities was less than 4600, while the vulnerabilities covered was almost 20,000 [7,8]. Figure 1 highlights the number of vulnerabilities throughout the years. During the last three years, vulnerabilities have peaked. Thus, leading to the importance of discovering and fixing the software vulnerabilities promptly [7].

In the past few years, cyber-attacks on both the industrial and commercial sectors have continued to rise. The "WannaCry" ransomware discovered in 2017 is one of the cyberattacks that had the worst impact, detrimental consequences, and the most comprehensive coverage in recent years. This ransomware affected 230000 computers globally. The most notable of the institutions affected was the UK National Health Service, which suffered £92 million in losses. The total cost of damages from "WannaCry" resulted in $ 4 billion in losses[9]. Vulnerability identification is a challenging problem in security. Besides the use of classical approaches throughout the years, such as static analysis[10,11,12,13,14], dynamic analysis[10,13,2] or hybrid approaches[10], several advances in the application of machine learning as a complementary have been made. Several early methods included the use of features or patterns hand-crafted by human experts as input to machine learning algorithms to detect vulnerabilities. However, due to the variety in root causes based on the weaknesses[5], and libraries, the characterization of all the vulnerabilities from numerous libraries is impractical when using hand-crafted features.

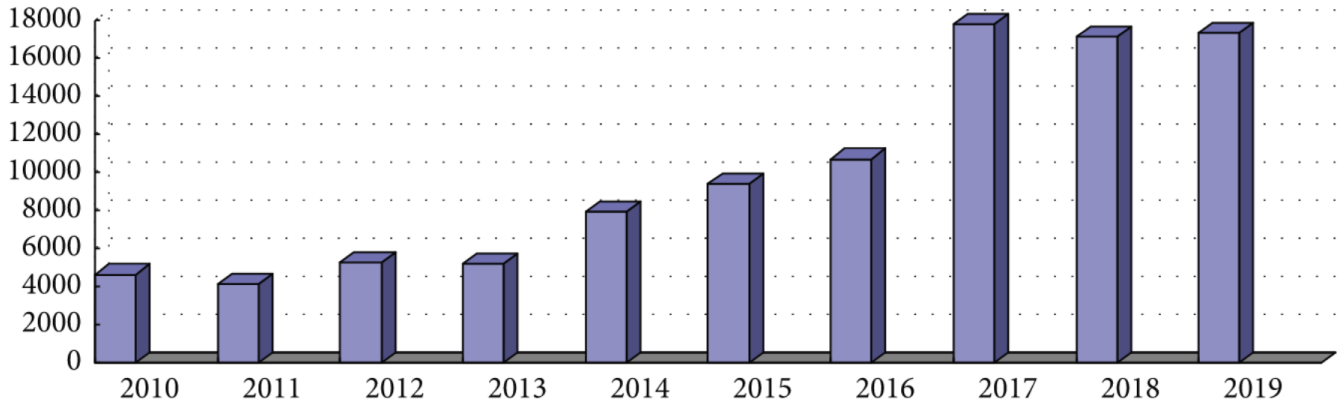

FIGURE 1 Number of CVE vulnerabilities over the years[7]

1.1 | Motivation

Machine Learning has been widely adopted in areas such as spam detection[15], and threat prediction[16]. With the rapid growth and development of neural networks, Deep Learning has been widely adopted in several areas of software security recently[17]. Some of the advantages of Deep Learning include avoiding the intense human work of experts as it requires no predefined rules. Source code of the software offers better readability for researchers as it contains rich syntactic and semantic information such as function calls, header files, and code modifications which can prompt the identification of hidden errors within the software.

Deep Neural networks have recently been extensively adopted in problems associated with natural language processing (NLP). Researchers have been seeking methods that involve combining deep neural networks with source code features for the automatic and rapid identification of vulnerabilities. Some researchers extract static features from source code and convert them to representations which are then used as inputs for neural networks. Due to the structure of the extracted features being presented like an NLP problem, researchers leverage deep learning models by treating source code as a natural language structure. For example, VulDeePecker[18] uses code gadgets to represent functions of the programs and leverages Long Short-Term Memory (LSTM) networks to discover vulnerabilities. Machine learning strategies for enhancing the performance of deep learning models have been growing in recent years. Due to the bias in representing vulnerable and non-vulnerable code and the need to improve the performance of vulnerability detection models, several machine learning strategies have been used to enhance deep learning models or modify the dataset used for the models. Despite the uses of machine learning strategies for improving deep learning models, to our knowledge, there has been no systematic evaluation that presents how effective the strategy would be when paired with source code representations.

Our Contributions: For this paper, we conduct a performance evaluation for source code vulnerability detection on deep learned features and evaluate how machine learning strategies can influence the model's performance. Our evaluation utilizes five source code representations, three deep learning models, and four machine learning strategies. The contributions of our experimental work are as follows: -

First, to experimentally show the effectiveness of implementing vulnerability detection, we collected the dataset used in five different works. Each of the datasets is representative of a source code representation. The datasets collected are the POSTER dataset [10], VulDeePecker dataset [11], Draper VDISC dataset [13], SySeVR dataset [12] and Devign dataset [8].

Second, we implement three Deep Neural Networks (DNNs), a Bidirectional Gated Recurrent Units (BGRU), a Bidirectional Long Short-Term Memory (BLSTM), and a Convolutional Neural Network (CNN) based on Yoon Kim's model [19]. These models are designed to be used as the base models for our project. The design of the DNNs is as follows: (i) The layers for BGRUs and BLSTMs are designed to include an embedding layer, a Bidirectional BGRU/LSTM Layer, GlobalMaxPooling1D, Dropouts, and Dense layers. (ii) The layers used for the CNNs consist of an input layer, embedding layer, reshape layer, Conv2D, Flatten, Dropout, Dense, and Concatenate layers. (iii) The activation function used for the models is SoftMax.

Third, we evaluate the quantitative impact of the Machine Learning (ML) strategies on vulnerability detection effectiveness on the datasets. Here modified models are created along with the machine learning strategies. The performance is then compared with the base model to evaluate how much the learning strategy influenced its performance.

Paper Organization: Section 2 introduces the background on the concepts and terms used for deep learning-based vulnerability detection. Section 3 describes the experimental setup, which consists of the details of the dataset and configurations. Section 4 presents the experimental results and analyses the answers to our research questions. Section 5 discusses the threats to the validity of our work. Section 6 reviews the works within the domain of our study. Section 7 summarizes the conclusion of our research and discusses potential future works.

2 | BACKGROUND

2.1 | Definition of vulnerabilities

Software vulnerabilities are bugs that affect security [5]. In short, vulnerabilities are a subset of software defects [20]. If system vulnerabilities are discovered, hackers can easily exploit them, leading to security breaches due to policy violations [10]. Software vulnerabilities are defined as bugs arising from flaws discovered during the design, development, or modification of software that could be used to violate certain or implicit types of security policies.

2.2 | Categories of vulnerability detection

Vulnerable code pattern based: This ML-based vulnerability discovery category includes supervised ML-based approaches that filter vulnerable code patterns from numerous vulnerable code examples. Then it uses a pattern matching method to detect vulnerabilities found in the source code. Text tokens extracted from the source code demonstrate this pattern (i.e., the bag of words method used to convert the source token to a vector [21]). Various techniques such as N-gram [22] are used to analyze the program source code. However, some researchers have argued that they rely solely on text mining strategies because they cannot understand the source code's semantic meaning. This failure was due to term co-occurrences, which illustrated counts and failed to maintain the code's contextual dependency [23]. Static code analysis tools have been used to extract structured features like code templates for ML algorithms [19]. Examples of these features include graph-based features such as Abstract Syntax Tree (AST), Control Flow Graph (CFG), Program Dependency Graph (PDG), Data Flow Graph (DFG), and more. Comparison with other software metrics and code frequency features shows more representation-based information generated by the features. For example, ASTs display the source code in a tree structure. On the other hand, PDG, DFG, and CFG analyze the flow of variables from source to target to understand program structure and dependencies. Yamaguchi et al. [24] combined AST, PDG, and CFG into a standard composition called Code Property Graphs (CPG) to provide a detailed graph structure for code attributes. Extracting advanced features based on the structure of source code can help identify vulnerabilities.

Software metrics-based: Several studies have used software metrics as feature sets to build models used for prediction. The model is then used to assess the vulnerability status of the software code based on these metrics. Measurements used include McCabe [25], and Code Churn [26], a numerical measure of software product quality from various perspectives. McCabe's metrics are used as software complexity metrics, whereas code churn indicates the likelihood that code changes frequently are error-prone. Despite the long history of anticipating vulnerabilities, it is not a direct indicator of a vulnerability, leading to several studies using this method to report poor detection results [10].

Anomaly based: Studies by Engler et al. [27] and Li et al. [28] used rules from people as a learning template for detecting vulnerable code detection without compliance with programming guidelines. Other studies used imports, function calls [28], and API usage patterns [27] to generate the relevant

vulnerability discovery features. These feature sets are associated with a small group of vulnerabilities and are limited in capacity. Such features include ones generated from imports or function calls[29] utilized for training the classifiers that locate vulnerabilities using header files and API usage symbols linked to absent checks for vulnerabilities. These absent checks result from the shortage of validation or boundary checks, which concluded that these studies' solutions might be relevant for task-specific applications.

2.3 | Machine Learning Strategies

Imbalance Handling: There arises a situation in vulnerability detection where the class distribution of the dataset has a skew problem. The class imbalance problem is a common problem that occurs in data mining[30]. In oversampling methods, minority class instances are increased to equate the classes. Common issues associated with oversampling methods include adding new instances to the dataset, which results in overfitting classifiers[31]. Amin et al.[32] used an oversampling method known as Synthetic Minority Oversampling Technique (SMOTE), which creates "synthetic" examples rather than replacing the minority class instances.

Attention Mechanism: The attention mechanism was first introduced for Natural Language Processing (NLP) tasks by Bahdanau et al.[33]. However, a similar idea called glimpses was proposed for computer vision by Larochelle, and Hinton[34], based on how biological retinas fixate on relevant parts of an optic array with resolution falling off rapidly with eccentricity. After a study by Mnih et al.[35], visual attention became very popular, which showed promise as it significantly outperformed state-of-the-art image classification tasks and dynamic visual control problems. For our experiment, we decided to use the attention mechanism in two ways. For the sequential neural network models, we use the Keras Self Attention package[36], and for the non-sequential neural network models, we use a custom attention layer inspired by Raffel et al.[37]. Besides offering a significant gain in performance, an attention mechanism can be used to interpret the behavior of neural architectures, which are challenging to understand. The knowledge gathered by neural networks is stored as numerical elements that by themselves do not provide means for interpretation. It then becomes difficult to pinpoint the reasons behind the wrong output of neural architecture. Attention has been viewed as a means that can provide the key to partially interpret and explain the neural network behavior[38], even though it is not considered as a reliable means for an explanation.

Transfer Learning: The idea behind transfer learning developed based on the phenomenon that deep neural networks trained on raw images learn similar features[39]. The initial layer features usually do not appear specific to a particular dataset or task but are general, making them applicable to many datasets and tasks. These standard features occur regardless of the dataset or cost functions, making them general and easily transferable for learning aspects of specific datasets. In transfer learning, the base network is first trained on the base datasets and tasks, and then the learned features are transferred to a target network to be trained on a target dataset or task.

Ensemble Models: Ensemble models are models that consist of a hybrid of different models that improve the models' sustainability. Ensemble models' candidate models share the same neural network structure but have various model checkpoints or different initial parameters. Although ensemble models with cross-validated unequal weights have a likeability to improve the robustness for noisy data, the deterioration caused by the change in the data distribution is inevitable in time series data as the ensemble weights are fixed[40].

Dimensionality Reduction: Dimensionality reduction involves reducing the number of input variables before a data mining algorithm can be applied successfully. Principal Component Analysis (PCA) is a dimensionality reduction method used to decrease the dimensions of large datasets by transforming large variable sets into smaller ones while containing most of the larger set[41]. Since more minor data makes it easier and faster for machine learning algorithms, it is easier to explore and analyze the data.

3 | EXPERIMENTAL SETUP

Framework: In data pre-processing, the first step involves parsing the source code into its relevant program representation for each of the datasets and generating pre-trained word2vec embeddings. The data is then tokenized, padded, shuffled, and divided into training and testing source code. We obtain the feature representations for the training set's classification models by converting the program representations into vectors while preserving the structural information[14]. Each vector representation is treated as its semantic structure, which maintains its meaning. The vectors extracted from the pre-processing phase are used as input for the neural network to train and evaluate whether the trained model can offer

acceptable detection performance. An overview of the framework is shown in Figure 2. For the implementation of the ML strategies, the base models' structures are modified to accommodate them.

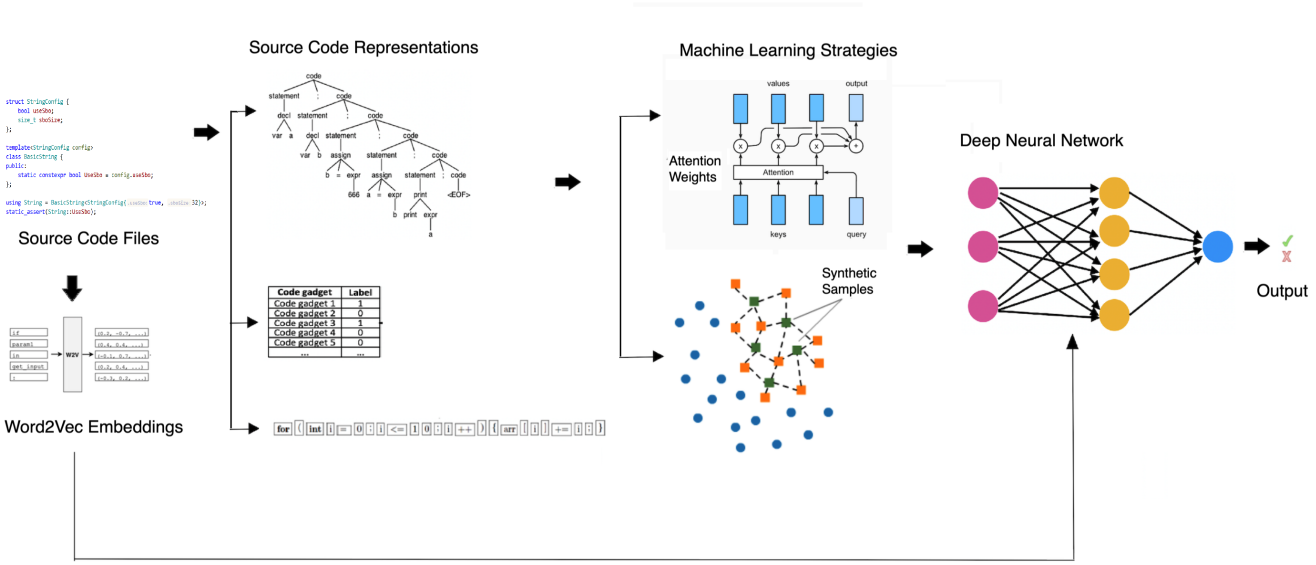

FIGURE 2 Framework for deep learning model

3.1 | Datasets

Our project aims to use datasets of code representations that are made publicly available or whose process of code representation generation could be replicated easily. This dataset shall be used to train our deep learning model. The main reason behind this choice for selecting this approach for generating source code representations is that it would be time-consuming and resource-heavy to generate the representations needed. In selecting the source code representations approach, several perspectives were considered—the main reason being which syntactic and semantic information could be captured from the source code. Liu et al.[42] argued on the necessity of AST representations to mine patterns from code while others such as Russel et al.[43] and Hovsepyan et al.[44] illustrate that this is not necessarily the case, as textual representations can also be used to model code. Furthermore, Dam et al.[45] argue that Abstract Syntax Trees (AST) may not be effective in capturing the semantic information deep in source code with human-engineered features and software metrics. We decided to select AST, Code Gadgets (CG), Lexed Code Representation from such perspectives. (LCR), Semantics-based Vulnerability Candidates (SeVC) and Composite Code Representation (CCR) as our source code representations. In our study, we collected the dataset used in five different works. Each of the datasets is representative of a source code representation that we chose. The datasets collected are the POSTER dataset[46] , VulDeePecker dataset[18], Draper VDISC dataset[43], SySeVR dataset[47]and Devign dataset[8]. We also create a combined dataset for each code representation. The dataset selected had a high level of data imbalance at a ratio of 90:10 for the non-vulnerable datasets and the vulnerable datasets.

After gathering all the data and filtering it, the total number of examples is summarized in Table 1. The dataset is divided into the total size of functions used for processing with the Deep Learning model. The total number of functions consists of both vulnerable and non-vulnerable functions. Our POSTER dataset consists of ASTs which were created from the projects FFmpeg, LibTiff, LibPNG, and VLC. For the VulDeePecker dataset, we decided to use the CGs comprising gadgets belonging to two vulnerability types, namely buer error vulnerability (CWE-119) and resource management error vulnerability (CWE-399). The Draper VDISC dataset consists of LCRs which belong to the vulnerability types CWE-119, CWE-120, CWE-469, CWE-476, and CWE-OTHER. The descriptions of what each CWE represents for the project are in Table 2. For the Devign dataset, we decided to use CCRs created from the projects FFmpeg and QEMU. For the SySeVr dataset, we decided to use the SeVCs, which involve vulnerabilities related to the below:

- Library/API function calls

TABLE 1 Dataset sizes

Code representation	Dataset name	Project name/ type	Dataset examples
AST	POSTER	FFmpeg	5112
		LibTiff	873
		LibPNG	542
		VLC	3672
CG	VulDeePecker	CWE119	39753
		CWE399	21885
LCR	Draper VDisc	CWE119	485140
		CWE120	46990
		CWE469	12065
		CWE476	2667
		CWEOTHER	398780
CCR	Devign	FFmpeg	9683
		QEMU	17549
SeVC	SySeVR	API	1735
		POINTERUSE	42262
		ARRAYUSE	276574
		INTEGEROVERFLOW	95353

TABLE 2 VDISC Draper dataset details

Draper VDISC dataset description	
CWE ID	CWE Description
CWE-119	Buffer Overflow
CWE-120	Improper Restriction of Operations within the Bounds of a Memory Buffer
CWE-469	NULL Pointer Dereference
CWE-476	Use of Pointer Subtraction to Determine Size
CWE-OTHER	Improper Input Validation, Useof Uninitialized Variable, Buffer Access with Incorrect Length Value, etc

- Arrays (e.g., improper use in array element access, array address arithmetic, address transfer as a function parameter)
- Pointers (e.g., improper use in pointer arithmetic, reference, address transfer as a function parameter)
- Improper arithmetic expressions (e.g., Integer overflows)

3.2 | Data Preprocessing

Abstract Syntax Trees (AST): ASTs are graph-based representations of functional aspects of source code such as expression, scopes, and declarations. ASTs are generated from source code on a class-based level. Extracting ASTs involves bypassing redundancies by eliminating unnecessary syntactic details, which do not affect the original code's function even after parsing. Exclusions include punctuation marks, commas, code comments, or whitespaces. To generate the AST representations, we use CodeSensor, a C/C++ parser built in Java. The ASTs are serialized and shown in a table format that depicts the nodes, depth, values, and type parsed from the code. When referring to AST nodes, we follow a convention by Lin et al.[46], which consists of three node types: i) placeholder nodes, ii) API nodes, and iii) syntax nodes. The serialized ASTs are then traversed using the depth-first traversal with each node and element mapped to a vector. For the ASTs, the POSTER dataset is used. This dataset consists of three open-source projects; FFmpeg, LibPNG, and LibTIFF. The data on the vulnerabilities are obtained from the National Vulnerability Database (NVD) and CVE. NVD and CVE use the CVE ID as the unique identifier for vulnerabilities. CVE IDs are assigned to vulnerabilities to help track their technical information. The ASTs are then labeled using their CVE IDs according to the process used by Lin et al.[46]. The functions with a CVE ID are labeled vulnerable, and functions without CVE IDs are labeled as not vulnerable. Due to the differences in naming conventions, the data must be normalized before training as the data is collected from different projects. All project-specific names are replaced under a universal naming convention done through textual mapping elements of vectors to numbers within a range. The mapping links each textual element of the vector directly to an integer, which automatically identifies each textual element. This method also works on maintaining semantic information. The vector α after normalization will be in the form [8, 9, 1, 17, 19, 83, 21, 14, 81, 15, 19, 83, 53, 59, 14, 83, 62, 54, 84]. The padding length of 1000 is chosen to create vectors of an equal length. Vectors longer than 1000 are truncated, and short vectors are padded at the end with zeros. The inspiration behind the selection of the number 1000 was based on the work of Lin et al.[48].

Code Gadgets (CG): A code gadget is a functional representation that comprises numerous program statements such as code lines that have a semantic relationship based on data or control dependency. The generation of code gadgets involves the use of methods within the source code. Library/API function calls are grouped into forward library/API function calls, and backward library/API function calls to extract the CGs. Forward library function calls receive one or more inputs directly from files or sockets. On the other hand, backward library function calls do not receive

external input from the environment they are executed. Program slices are then extracted from source code files based on API/Library function calls' arguments. We use the VulDeePecker dataset, CWE- 119 (SARD Buffer Error dataset), and CWE-399 (SARD Resource Management Error dataset).

Two types of slices are defined: forward slices and backward slices. The code gadgets are automatically labeled as' vulnerable' using "1" if it has a statement deleted or modified, or patched for data from NVD. It is labeled as' not vulnerable' using "0". The gadgets labeled as "1" are manually checked after automatic labeling for mislabeling. Files obtained from SARD are already labeled as good (if it contains no security issues) or bad (if it contains security issues), or mixed (if it contains functions with security issues and their patched versions). Gadgets extracted from a good program are labeled "0," whereas gadgets extracted from bad or mixed programs are labeled "1". If at least one vulnerable statement is contained, they are labeled as "0". Gadgets with conflicting labels (labeled as both "1" or "0") are eliminated. All the non- ASCII characters are eliminated, and user-defined variables and functions are mapped to symbolic names (i.e., "VAR1" and "FUN2"). The symbolic representation of the gadgets is divided into tokens by using lexical analysis. The tokens include keywords, operators, identifiers, and symbols. A fixed-length corresponding to a gadget is set to the value k to ensure the vectors created are equal. When a vector is shorter than k, backward slices of the code gadget are generated. The value of k for our study is set to 50.

Semantics-based Vulnerability Candidates (SeVC): SeVCs are source code representations inspired by regions in image processing source code is divided into smaller pieces of code at the function level, which may exhibit syntax and semantic characteristics of vulnerabilities. In terms of the SeVCs used in this project, we follow the procedure by Li et al.[47]. The first step is the extraction of Syntax-based Vulnerability Candidates (SyVCs). These are code elements that may or may not be vulnerable based on syntax characteristics of known vulnerabilities. SyVCs are the start point for the extraction of SeVCs and contain an ordered set of tokens that include operators, keywords, etc. For the extraction of SyVCs, the program is first converted into ASTs using Joern. The AST is then traversed to find the SyVCs concerning vulnerability syntax characteristics. Then, the CheckMarx tool is used to define vulnerability syntax characteristics. The checkmark rules provided four primary vulnerability characteristics Array Usage, Library/API Function Call, Pointer Usage, and Arithmetic Expression. For our experiment, the SySeVR dataset, which consists of the functions of source code projects representing the four primary vulnerability characteristics, is used. The next step is to convert SyVCs to SeVCs. SeVCs contain various statements semantically related to the SyVC based on control dependency, data dependency, or both. Joern does the extraction of PDGs for each SyVC to use the program slicing technique to identify semantically related statements based on data dependency and control dependency. For each program function, a PDG is created. Each PDG consists of a CFG of the function and a set of direct edges where each edge represents data or control dependency that flows between a pair of nodes. Forward and backward slices are generated from each element to create program slices. These slices are merged to create program slices. The statements that belong to a function in a program slice appear as nodes to a SeVC. They are then transformed while preserving the order of the function's statements. The next step is changing statements that belong to different functions into SeVCs. SeVCs from the NVD are labeled upon analysis of the diff files of vulnerabilities with line deletions. If diff files contain at least two deletions or modified statements prefixed with "-," the SeVC is labeled "1," which illustrates that it is vulnerable. Besides, if the diff files have at least one moved statement that begins with "-" and is known to have a vulnerability, it is also labeled "1". If it does not meet the criteria mentioned above, the SeVC is labeled as "0," which illustrates that it is not vulnerable. For the SeVCs obtained from the SARD dataset, SeVCs collected from "good" programs are labeled as "0," while those from "mixed" or "bad" programs that contain at least one vulnerability are labeled as "1". To create vectors of equal length, the fixed length, the length chosen, is set to 500. If a vector is shorter than 500, zeros are padded onto the end. Likewise, if a vector is longer than 500, three scenarios are considered. (1) if the portion of a vector corresponding to a forward slice is shorter than 500, then the vector's leftmost part is excluded. (2) if the portion of a vector corresponding to a backward slice is shorter than 500, then the vector's rightmost part is excluded. (3) Otherwise, the same length is deleted from both sides to create a vector of the length 500.

Lexed Code Representation (LCR): Source code lexing involves the extraction of useful features from the raw source code of each function. For the Lexed code representation, The lexer designed by Russell et al.[43] reduced the code representations to having a total vocabulary size of 156 tokens. The vocabulary includes all the base C/C++ keywords, operators, and separators. All codes which do not affect compilation or removed (i.e., comments). Types such as String, character, and float literals are all lexed to type-specific placeholder tokens. The digit tokenizes integer literals as these values are highly relevant to vulnerabilities. Types and function call from standard libraries with a high likeability to getting vulnerabilities are mapped to generic versions (i.e., u32, DWORD, uint32t, uint32, UINT32 and are all lexed to be the same generic token representing 32-bit unsigned data type). These individual tokens' learned embeddings are distinguished based on the code they commonly use, thus building the desired invariance. Our experiment uses the Draper VDISC dataset, which consists of the projects shown in Table 2. For the datasets made available by the authors, we first create a new data frame from the HDF5 file. Then the columns are renamed to match the CWE-ID and function source column. Then the data frames representing the training, test, and validation data are serialized into pickle files. Subsequently, word-level tokenization is conducted on the texts. After that, sequence files are then created from the tokens. We use the embedding dimension size of 13 with five outputs based on the dataset's structure for our embedding layer. Each of the outputs corresponds to the binary figures of "1" for being

vulnerable and "0" for being not vulnerable.

Composite Code Representation (CCR): Concerning the composite code representation, The idea behind it stems from the fact that various program representations manifest more profound semantic knowledge behind textual code in program analysis. Classical graph-based models such as ASTs, CFGs, and Dataflows capture both the syntactic and semantic relationships among different source code components. However, according to Yamaguchi et al.[24], some vulnerability types, such as memory leaks, are too complex to be detected without a collective consideration of the composite code semantics. ASTs alone have several limitations and can be used to find only arguments deemed insecure. The combination of ASTs and CFGs allows for higher coverage as two more vulnerability types can be accommodated (i.e., resource leaks and use-after-free vulnerabilities). The three code graphs can describe most vulnerabilities through further integration, excluding those requiring extra external information (i.e., race condition and design errors requiring significant detail). It is evident that manually crafted vulnerability templates in a graph traversal form convey key insights and prove the feasibility of learning broader vulnerability patterns by integrating properties belonging to ASTs, CFGs, and DFGs into a joint structure. Another code structure known as Natural Code Sequence (NCS) is also considered since several studies related to machine learning have demonstrated its effectiveness[18,43]. Its unique flat structure complements the classical representations and captures code tokens' relationships in a human-readable manner. The ASTs, DFGs, NCSs, and CFGs are combined into one graph to create the composite code representation. For the datasets that were made available by the authors[8]. First, we create a new data frame from the files and extract the columns "func," which represents the function, and "target," which means the status of the vulnerability, "1" for vulnerable and "0" for not vulnerable. The data is then padded through 'post' padding and split into train, test, and validation with a ratio of 80:10:10, respectively. Regarding the embedding dimension used, the size was 100.

Concerning the composite code representation, The idea behind it stems from the fact that various program representations manifest more profound semantic knowledge behind textual code in program analysis. Classical graph-based models such as ASTs, CFGs, and Dataflows capture both the syntactic and semantic relationships among different source code components. However, according to Yamaguchi et al.[24], some vulnerability types, such as memory leaks, are too complex to be detected without a collective consideration of the composite code semantics. ASTs alone have several limitations and can be used to find only arguments deemed insecure. The combination of ASTs and CFGs allows for higher coverage as two more vulnerability types can be accommodated (i.e., resource leaks and use-after-free vulnerabilities). The three code graphs can describe most types of vulnerabilities through further integration, excluding those requiring extra external information (i.e., race condition and design errors requiring significant detail). It is evident that manually crafted vulnerability templates in a graph traversal form convey key insights and prove the feasibility of learning broader ranges of vulnerability patterns by integrating properties belonging to ASTs, CFGs, and DFGs into a joint structure. Another code structure known as Natural Code Sequence (NCS) is also considered since several studies related to machine learning have demonstrated its effectiveness[18,43]. Its unique flat structure complements the classical representations and captures code tokens' relationships in a human-readable manner. The ASTs, DFGs, NCSs, and CFGs are combined into one graph to create the composite code representation. For the datasets that were made available by the authors[8]. First, we create a new data frame from the files and extract the columns "func," which represents the function, and "target," which means the status of the vulnerability, "1" for vulnerable and "0" for not vulnerable. The data is then padded through 'post' padding and split into train, test, and validation with a ratio of 80:10:10, respectively. Regarding the embedding dimension used, the size was 100.

3.3 | Model Architectures

For the three model types selected for use on the code representations, we created sequential and non-sequential models using the Keras library. Each model with different layers to varying degrees based on the code representation. Below are the architectural structures of the models used.

BLSTM model: For our BLSTM model, we designed it using the Keras library. The common layers used on our base BLSTM models include an embedding layer, a Bidirectional LSTM Layer, GlobalMaxPooling1D, Dropouts, and Dense Layers. The BLSTM model used for the base model of our ASTs, CGs, SeVCs, and CCRs, is sequential. The model used for the Lexed Code Representation is non-sequential.

BGRU model: BGRU Model: For our BGRU model, we designed it using the Keras library. The design was similar to the BLSTM Model. The common layers used on our base include an embedding layer, a Bidirectional LSTM Layer, GlobalMaxPooling1D, Dropouts, and Dense Layers. BGRU Model used for the base model of our ASTs, CGs, SeVCs, and CCRs is sequential. The model used for the LCR is non-sequential.

CNN Model: For our CNN Model, we designed it using the Keras library. The design we implemented was based on Yoon Kim's CNN model[49] with some modifications on some of the layer inputs, which consists of an input layer, embedding layer, reshape layer, Conv2D, Flatten, Dropout, Dense and Concatenate. This model was used on the CNNs for the ASTs, CGs, SeVCs, and CCRs. However, for the LCRs, another CNN model

which consists of a Convolution1D layer is used along with MaxPool1D, Dropout, Flatten, and Dense Layers.

3.4 | Word2Vec Embeddings

To encode the tokens of the source code, we decided to use word2vec. The due to the unavailability of word2vec models, we decided to train our models against our source code functions, using their tokens as a part of the corpus. The Hyperparameters used for the word2vec models include the dimensionality of the resulting vectors and the minimum times that a token appears in a corpus to be included in the model. Several hyperparameter settings were tested and evaluated to see how useful the trained embedding was. Our two approaches for evaluating the trained embeddings' usefulness included (1) Simply looking at the tokens and the most similar tokens according to the model to see if they make sense. This is more of a subjective approach. (2) Evaluating the embeddings based on the final performance of the base models used. This approach works well only if the embeddings on the training data are sensible. The parameters we decided to use for our word2vec model are the size of 100 and 50 iterations.

3.5 | Machine Learning Strategy Implementation

Imbalance Handling: For our experiment, we decided to use SMOTE as our imbalance handling learning strategy for oversampling on the datasets before using them as inputs for our DNNs. SMOTE was applied to the POSTER dataset, VulDeePecker dataset, SySeVR dataset, and Devign dataset. Due to difficulties in the implementation, the Draper VDISC dataset was excluded from the imbalance handling experiment using SMOTE. Imbalance handling is applied to all the Deep Learning models used.

Attention Mechanism: In terms of implementing the attention mechanism, we decided that the mechanism would be applied in two ways. For the sequential CNN models used with the POSTER dataset, VulDeePecker dataset, SySeVR dataset, Draper VDISC dataset, and Devign dataset, we decided to use the attention scheme for this experiment is Sequence-Self-Attention, which works by processing sequential data that considers the context of each timestamp. The sequence self-attention was implemented through the Keras library SeqSelfAttention [36]. For the non-sequential models used with the POSTER dataset, VulDeePecker dataset, SySeVR dataset, Draper VDISC dataset, and Devign dataset, the attention mechanism that we decided to use was inspired by the model proposed by Raffel et al. [37]. A custom attention layer is designed based on a single vector's production from an entire sequence in this model. This attention model was implemented by adding the custom attention layer after the RNN or Conv layer with the return sequences set as "True." The output dimensions are inferred based on the output shape of the model. Attention is applied to all the Deep Learning models used.

Transfer Learning: To conduct transfer learning, we decided to implement cross-project transfer learning to be implemented between the source code representations' projects. The main inspiration of how we conducted transfer learning was based on Lin et al. [48]. The idea was to train a model with a set of projects and gauge whether transferred knowledge can train another project. Due to limitations such as the dataset sizes and complexities, we decided to conduct transfer learning on the Poster dataset and SySeVR dataset. The other projects were excluded from the transfer learning experiment. For the POSTER dataset, the projects used to train the model to combine the FFmpeg, LibPNG, and LibTIFF. The target project is the VLC project. Besides, for the SySeVR dataset, the project used to train the model combines the ARRAYUSE, POINTERUSE, and INTEGEROVERFLOW project. The target project is the API project. The models we used for transfer learning were the BLSTM, BGRU, and CNN models.

Ensemble Models: The ensembled models that were decided for this experiment involve combining the CNN, LSTM, and GRU models that we used. The idea was to determine whether using a combination of models that use outputs learned from previous models in a sequence can improve the model performance. Due to practical considerations, we decided to design our ensemble model using CNN, BLSTM, and BGRU. However, the number of layers used for the BLSTM and BGRU was decreased to 32 layers. The decrease in layers was due to the amount of time for training with our computer configuration. Ensemble models are used for all the datasets.

Dimensionality Reduction: For dimensionality reduction, we decided to use Principle Component Analysis (PCA). To implement PCA, we decided to normalize the data to reduce the data dimension and define the number of components to which the dimensions were reduced. Then, we transformed the training and testing data and implemented it with our models. Due to difficulties in the implementation, the Draper VDISC dataset was excluded from the PCA. However, PCA was implemented on all the other representations. The number of components for PCA for

the VulDeePecker dataset is 30, and 100 for the POSTER dataset, SySeVR dataset, and Devign dataset.

3.6 | Practical considerations

Our models are implemented using Python and trained using Keras with the Tensorflow backend. The machines used to run our experiment are a Windows 10 PC with an NVIDIA Quadro P2000 GPU and an Intel Xeon Silver 4110 CPU operating at 2.10GHZ. The two hyperparameters for this study are optimizers and batch size. The main optimizers chosen are Adam and Adamax. For the training batch sizes, we considered three sizes (i.e., 32, 64, 128). The number of training epochs was set to the arbitrary value of 150. We monitor the validation loss during the training and use early stopping to stop training when the lowest validation loss is obtained. The data is partitioned into mutually exclusive training, testing, and validation sets. 80% of the data is randomly selected as the training set, 20% for the testing (which is further split to 10% for testing and 10% for validation). This is significantly in line with the general practice in training neural networks and other works with similar vulnerability detection objectives. For example, Russell et al.[43] split their dataset into 80% training, 10% validation, and 10% testing, and Li et al.[18] used a data split of 80% and 20% for training and testing, respectively. It is important to note that the validation set is not used to learn any parameter but rather to evaluate the model performance after its parameters are learned from the training set.

3.7 | Evaluation metrics

Four key areas usually form the basis for evaluation for the effectiveness of our models: true positives, true negatives, false positives, and false negatives. Two metrics are directly derived from these values: precision and Recall. Precision is the rate of true positives within the positives. And Recall, known as the sensitivity or the measurement of the rate of positives correctly identified compared to the total actual positives. For this experiment, the chosen evaluation metrics shall be based on the F1 score (F1) and Accuracy (ACC) similar to that of the work of Zhou et al.[8]. The equation shall be based on letting TP be the total of vulnerable samples detected correctly. FP is the total of false vulnerable samples. FN be the total of true undetected vulnerable samples, and TN the total of non-vulnerable samples detected.

Accuracy: This can be defined as the rate of the total number correct predictions in comparison to the total number of predictions.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision: This measures the number of correct vulnerable sample predictions out of the total number of vulnerable samples predicted.

$$Precision = \frac{TP}{TP + FP}$$

Recall: This measures the number of correct vulnerable samples predictions out of the actual total number of vulnerable samples present.

$$Recall = \frac{TP}{TP + FN}$$

F1-score: This is the harmonic mean of both precision and recall. This value measures the overall effectiveness of the model.

$$F1 - score = \frac{2PR}{P + R}$$

4 | EXPERIMENTAL RESULTS AND ANALYSIS

For this study, we focused on studying the effects of the machine learning strategies when paired with source code representations. Each of the datasets consists of different representations constructed from various datasets with different granularity and construction methods. These factors have significance in the performance of DL-based vulnerability detection. Furthermore, the datasets are collected from various sources. For example, some of the data constructed are synthetic data, semi-synthetic data, and real data. To evaluate the effectiveness of the ML strategies on our datasets, we focus on five research questions:

- Which Machine Learning Strategy provides the best performance improvement when paired with the POSTER dataset?
- Which Machine Learning Strategy provides the best performance improvement when paired with the VulDeePecker dataset?
- Which Machine Learning Strategy provides the best performance improvement when paired with the SySeVR dataset?

TABLE 3 Poster dataset experimental results

Learning strategy	DL model	FFmpeg		LibPNG		LibTiff		Combined		VLC	
		f1-score	accuracy	f1-score	accuracy	f1-score	accuracy	f1-score	accuracy	f1-score	accuracy
Base Score	BGRU	nan	0.957831	nan	0.941605	nan	0.872093	0.111111	0.956492	nan	0.990573
	BLSTM	nan	0.957831	nan	0.941605	nan	0.872093	0.111111	0.956492	nan	0.990573
	CNN	nan	0.957831	nan	0.941605	nan	0.872093	nan	0.954452	nan	0.990573
Dimensionality Reduction	BGRU	nan	0.957831	nan	0.941605	nan	0.872093	nan	0.954452	-	-
	BLSTM	nan	0.957831	nan	0.941605	0.226804	0.127906	nan	0.954452	-	-
	CNN	nan	0.957831	nan	0.941605	nan	0.872093	nan	0.954452	-	-
Attention	BGRU	nan	0.957831	nan	0.941605	0.370370	0.901162	0.111111	0.956492	-	-
	BLSTM	nan	0.957831	nan	0.941605	0.370370	0.901162	0.111111	0.956492	-	-
	CNN	nan	0.957831	nan	0.941605	nan	0.872093	nan	0.956492	-	-
Ensemble model	Ensemble network	nan	0.957831	nan	0.941605	nan	0.872093	nan	0.954452	-	-
Imbalance handling	BGRU	nan	0.957831	nan	0.941605	nan	0.872093	nan	0.954452	-	-
	BLSTM	nan	0.957831	nan	0.941605	nan	0.872093	nan	0.954452	-	-
	CNN	nan	0.957831	nan	0.941605	nan	0.872093	nan	0.954452	-	-
Transfer Learning	BGRU	-	-	-	-	-	-	-	-	nan	0.990573
	BLSTM	-	-	-	-	-	-	-	-	nan	0.990573
	CNN	-	-	-	-	-	-	-	-	nan	0.990573

TABLE 4 VulDeePecker dataset experimental results

Learning strategy	DL Model	CWE-119		CWE-399		Combined	
		f1-score	accuracy	f1-score	accuracy	f1-score	accuracy
Base Score	BGRU	0.857515	0.931329	0.948299	0.965273	0.906806	0.945165
	BLSTM	0.864109	0.932712	0.952447	0.968700	0.901983	0.944273
	CNN	0.805563	0.910325	0.935236	0.957505	0.844492	0.918802
Dimensionality Reduction	BGRU	0.476104	0.782165	0.643045	0.721498	0.635261	0.785204
	BLSTM	0.359773	0.772607	0.748850	0.862691	0.659197	0.822355
	CNN	nan	0.739152	0.573084	0.764450	0.044141	0.715444
Attention	BGRU	0.860578	0.931455	0.949165	0.967329	0.907344	0.94914
	BLSTM	0.865687	0.933342	0.947518	0.966187	0.904983	0.946788
	CNN	0.774605	0.897749	0.923457	0.950423	0.792429	0.893251
Ensemble model	Ensemble network	0.868202	0.932336	0.949705	0.966872	0.903207	0.944922
Imbalance handling	BGRU	0.445399	0.350396	0.691482	0.704592	0.489017	0.396171
	BLSTM	0.51592	0.510502	0.784143	0.818369	0.519015	0.464471
	CNN	0.41422	0.262231	0.569523	0.498515	0.448502	0.289422

- Which Machine Learning Strategy provides the best performance improvement when paired with the Draper VDISC dataset?
- Which Machine Learning Strategy provides the best performance improvement when paired with the Devign dataset?

The purpose of RQ1 to RQ5 is to identify which machine learning strategy has the most impact on each of the datasets representing a source code representation. Each machine learning strategy provides some form of positive or negative gain on the evaluation metrics when implemented. Thus, given the base scores of the datasets when used with the Neural Network models, we want to analyze to what extent there would be a gain or loss in the base score metrics when the ML strategies are implemented on the models.

The scores for each of the source code representations will be recorded in a table with color. The significance of the colors is as follows:

- Blue indicates the base model score or the result of the learning strategy being the same as the base model score.
- Red indicates that the score achieved is lower than that of the base model score
- Green indicates that the score is higher than that of the base model score

TABLE 5 VDISC Draper dataset experimental results

Learning strategy	DL model	CWE-119		CWE-120		CWE-469		CWE-476		CWE-OTHER	
		f1-score	accuracy	f1-score	accuracy	f1-score	accuracy	f1-score	accuracy	f1-score	accuracy
Base Score	BGRU	0.370122	0.960243	0.420197	0.922801	0.029851	0.99796	0.32	0.989598	0.336176	0.947488
	BLSTM	0.327704	0.962212	0.411342	0.922966	0	0.998023	0.255268	0.990296	0.298455	0.962236
	CNN	0.467783	0.972332	0.480385	0.93984	0.043321	0.997921	0.349585	0.990163	0.387541	0.960204
Attention	BGRU	0.190965	0.974011	0.409496	0.928779	0	0.998023	0.175155	0.989582	0.247691	0.96102
	BLSTM	0.218846	0.972167	0.410954	0.920864	0	0.998023	0.080495	0.990681	0.206762	0.965029
	CNN	0.258277	0.962919	0.412542	0.922542	0	0.998015	0.244386	0.988649	0.336418	0.952744
Ensemble model	Ensemble network	0.435874	0.97143	0.475587	0.942876	0	0.998023	0.404643	0.988735	0.374496	0.958596

TABLE 6 SYSeVR dataset experimental results

Learning strategy	DL model	API		ARRAYUSE		POINTERUSE		INTEGEROVERFLOW		Combined	
		f1-score	accuracy	f1-score	accuracy	f1-score	accuracy	f1-score	accuracy	f1-score	accuracy
Base Score	BGRU	0.903213	0.961649	0.901951	0.95122	0.914036	0.983484	0.900914	0.968179	0.917199	0.978869
	BLSTM	0.89893	0.960407	0.904627	0.948023	0.915141	0.983964	0.908441	0.971113	0.912687	0.977953
	CNN	0.804754	0.922211	0.840118	0.916765	0.732578	0.952269	0.837725	0.94719	0.793168	0.944237
Dimensionality Reduction	BGRU	0.427023	0.737287	0.45509	0.784513	nan	0.902791	0.469157	0.862108	0.06583	0.869302
	BLSTM	0.406773	0.755221	0.40169	0.782027	nan	0.902791	0.502895	0.864365	0.015729	0.868085
	CNN	nan	0.791398	nan	0.740114	nan	0.902791	nan	0.842699	nan	0.867081
Attention	BGRU	0.895023	0.958078	0.898641	0.946128	0.914045	0.983896	0.910384	0.971564	0.919885	0.979409
	BLSTM	0.900823	0.960717	0.892551	0.942103	0.911221	0.983125	0.918596	0.975401	0.918301	0.979095
	CNN	0.800149	0.916932	0.785247	0.891073	0.7533	0.952286	0.763126	0.912435	0.767492	0.938039
Ensemble model	Ensemble network	0.904949	0.962425	0.905483	0.947549	0.904359	0.981994	0.91523	0.973369	0.914384	0.978154
Imbalance handling	BGRU	0.346441	0.212949	0.412594	0.260005	0.177193	0.097209	0.280539	0.193184	0.23465	0.13292
	BLSTM	0.346143	0.212794	0.421953	0.287946	0.177196	0.097226	0.281952	0.199954	0.234649	0.132919
	CNN	0.345195	0.208601	0.412555	0.259886	0.177193	0.097209	0.271840	0.157301	0.234649	0.132919
Transfer Learning	BGRU	0.011364	0.842925	-	-	-	-	-	-	-	-
	BLSTM	nan	0.842699	-	-	-	-	-	-	-	-
	CNN	0.755906	0.923042	-	-	-	-	-	-	-	-

TABLE 7 Devign dataset experimental results

Learning strategy	DL model	FFmpeg		QEMU		Combined	
		f1-score	accuracy	f1-score	accuracy	f1-score	accuracy
Base Score	BGRU	0.551653	0.555783	0.541521	0.625641	0.598615	0.596999
	BLSTM	0.594877	0.562948	0.589971	0.603989	0.593848	0.596449
	CNN	0.580871	0.571648	0.555556	0.614815	0.551832	0.579063
Attention	BGRU	0.539149	0.551177	0.56305	0.617949	0.583254	0.59828
	BLSTM	0.581485	0.555783	0.551377	0.619373	0.560897	0.598829
	CNN	0.417231	0.515353	0.557264	0.598006	0.565914	0.588397
Ensemble model	Ensemble network	0.646531	0.556807	0.557905	0.624786	0.619958	0.599927
Imbalance Handling	BGRU	0.542916	0.547595	0.597174	0.634473	0.54958	0.60761
	BLSTM	0.586393	0.567554	0.529116	0.626781	0.545304	0.598646
	CNN	0.571283	0.570624	0.559765	0.615954	0.569324	0.582723
Dimensionality Reduction	BGRU	0.659729	0.497953	nan	0.582906	0.297484	0.534956
	BLSTM	0.659729	0.497953	nan	0.582906	nan	0.54063
	CNN	0.664388	0.497441	nan	0.582906	nan	0.54063

4.1 | RQ1: Which Machine Learning Strategy provides the best performance improvement when paired with the POSTER dataset?

To answer RQ1, we study the effects of the machine learning strategies on the f1-score and accuracy recorded from the base scores and models. The F1-score and accuracy of each project/example are compared with that of the base model. We compute the average performance of the Deep Learning models when paired with the Machine Learning strategies. The results calculated for the learning strategies with the POSTER

dataset can be found in Table 3.

Upon analyzing the base model scores, we found that the f1-scores of the Deep Learning models presented us with a score of “nan” for all the projects, excluding the combined dataset for the BGRU and BLSTM. For the CNN, the f1-score presented was “nan” for all the projects. The score “nan” reflects that the f1-score is undefined for reasons such as the model having the precision value defined as “nan,” showing that the value of true and false positives was “0”. We found no change in the f1-score and accuracy for the FFmpeg, LibPNG, and the combined dataset to apply the Attention mechanism. However, for the BGRU and BLSTM, the LibTiff project experienced an increase in both the f1-score and the accuracy. The increase was 37% and 3.9%, respectively. However, for CNN, the scores for all the projects remained unchanged. The increase in the f1-score and accuracy may have resulted from the attention mechanism interpreting some semantic correlation from the LibTiff project ASTs.

For the PCA application, there was no change in the f1-score and accuracy for the CNN. The BLSTM and BGRU reported a shift in the f1-score, which was “nan” for the combined dataset. A decrease in the accuracy and an increase in the f1-score of the LibTiff project of the BLSTM were also reported. The increase in the f1-score was by 22.6%, and the decrease in the accuracy was by 74.4%. This occurrence could be due to the dimensions of the dataset decreasing, thus leading to a precision of 12.7% and the model being unable to identify any false negatives.

For the application of our ensemble model, to determine which model performed best with a source code representation, we based it on whether the model has a better score than all the other base models projects (i.e., the ensemble model score is higher than the CNN, BGRU, and BLSTM score). The reason behind this is to illustrate whether a combined model can perform better than the best individual model score for a particular project. The score for all the projects remained the same, excluding the combined datasets, which decreased the accuracy by about 0.2%. The reason for the decrease in the accuracy could be due to the decreased weights used for the layers of the ensemble model. However, for the rest of the projects, the scores remained unchanged.

For the application of SMOTE, we found that all three models reported the same f1-score and accuracy. Compared to the base model, we find that for the BLSTM and BGRU, there was a change in the f1-score as the result presented were “nan” for the combined dataset and the accuracy had a decrease of 0.2%. We found that the accuracy and f1-score were not influenced for the transfer learning scenario and remained unchanged. The main reason could be the model not learning anything from the base layers that were transferred. The above results do not satisfy our expectations. We believe the Deep Neural Networks either had a result of “nan” in their f1-scores or had a decrease in the accuracy because the method used to process the ASTs may not have been effective, thus losing the syntactic meaning of the source code processed. Though for the LibTiff project, there was an increase in the accuracy and f1-score for the BLSTM and BGRU, there was still no significant change in the results of the other projects as the scores remained the same.

In summary, the answer for RQ1 is that the attention mechanism provided the most gains on the neural networks for Deep Learning-based vulnerability detection.

4.2 | RQ2: Which Machine Learning Strategy provides the best performance improvement when paired with the VulDeePecker dataset?

To answer RQ2, we study the effects of the machine learning strategies on the f1-score and accuracy recorded from the base scores and models. The VulDeePecker dataset consists of vulnerable and non-vulnerable examples of the CWE119 and CWE399 vulnerability types. In terms of the base model scores, we found that the accuracy and f1-scores were significantly high for the BLSTM and BGRU but were slightly lower for the CNN model. The results computed for the learning strategies with the VulDeePecker dataset can be found in Table 4.

Upon applying the attention mechanism, we found that the BGRU experienced a slight gain in the f1-scores of all the examples under the vulnerability types CWE119, CWE399, and Combined dataset 0.3%, 0.086%, and 0.05%, respectively. The accuracies of the same examples improved by 0.0125%, 0.2%, and 0.39%, respectively. However, for the BLSTM, the attention mechanism’s application improved the f1- scores for the CWE119 and Combined dataset examples by 0.1577% and 0.1824%, respectively, and the accuracy improved 0.0628% and 0.2514%, respectively. For the remaining examples, there was a slight decrease in the f1-score and accuracy. The application of attention on CNN did not have a positive effect on the metric scores. For the CNN, the attention mechanism did not prove very effective could be due to the limitation of the CNN due to the manner that the CNN SeqSelf-Attention layer may compute the relevant aspects of the source code between its input elements and the correlation the convolution layers.

For the application of PCA, we found that the f1-scores and accuracies of all the projects and neural networks decreased drastically compared to the base model. The average drop in the accuracy and f1-score for all the models was 20% and about 30%, respectively. This is due to the dimensions of the data being decreased, thus leading to poor performance.

For the application of the ensemble model, we wanted to identify if the model had higher accuracies and f1-scores for the dataset compared to the highest base model scores. We found that for the dataset correlating to the CWE119 vulnerability type, the f1-score slightly increased by 0.041%. However, for the other datasets, the scores were less than the best-performing base model scores. The reported scores might have been due to the CNN model having a slightly worse performance which might have influenced the output results of the ensemble model. Conversely, upon applying SMOTE to our dataset and creating synthetic examples, we found that the performance of all the models decreased drastically. The reason could be due to a lower rate of false-negative examples found by the three models.

In summary, the answer for RQ2 is that the attention mechanism provided the most gains on the neural networks for Deep Learning-based vulnerability detection. the main reason for its selection was attention being the only learning strategy that provided a positive increase in the accuracy and f1-scores of the models.

4.3 | RQ3: Which Machine Learning Strategy provides the best performance improvement when paired with the SySeVR dataset?

For RQ3, we study the effects of the machine learning strategies on the SySeVR dataset. We look at the impact of the machine learning strategies on the f1-score and accuracy recorded from the base scores and models. The VulDeePecker dataset consists of vulnerable and non-vulnerable examples of the API, ARRAYUSE, POINTERUSE, and INTEGEROVERFLOW vulnerability categories. The dataset also contains a combined dataset consisting of ARRAYUSE, POINTERUSE, and INTEGEROVERFLOW vulnerability categories. The results computed for the learning strategies with the SySeVR dataset can be found in Table 6.

We found a slight decrease in the accuracy and f1-score of the API and ARRAYUSE examples dataset with a slight decline of 0.1% on the f1-score and accuracy for the attention mechanism application for the BGRU model. The reason behind this might have been because of the dataset sizes being smaller. However, there was a slight increase for the POINTERUSE, INTEGEROVERFLOW, and combined dataset with an average gain of 0.002% for the f1-score and accuracy. This slight increase could be a result of attention faring better on the larger dataset size. For the BLSTM, the dataset under the API, INTEGEROVERFLOW, and combined dataset vulnerability category had an average increase of 0.1% for the f1-score and 0.003% for the accuracy. The only dataset that had a slight gain in the f1-score and accuracy of the CNN was the dataset under the POINTERUSE vulnerability category with an increase of 0.2% for the f1-score and 0.001% for the accuracy%. The reason behind the decline in performance for the CNN could be how the SeqSelf-Attention layer interprets the relevant aspects of source code between the input elements, which may have led to a decline in performance.

For the application of PCA, we found that all the datasets suffered a significant loss of approximately 15% for their accuracy and 40% for the f1-scores. The reason behind this drastic loss could be due to the dataset losing its semantic relation due to a decrease in the dimensions of the data. As expected, through implementing PCA on the dataset, the model's performance drastically decreased, and we found that the model training time was much faster than that of the base model. For the CNN, the f1-scores of all the datasets were "nan" because of the precision value being "nan." When compared against the best base model scores for each project, the ensemble model had a slight gain in the f1-score and accuracy with a value of 0.1% for the API vulnerability category dataset. There was also an increase in the f1-score by 1% for the INTEGEROVERFLOW vulnerability category dataset.

We used the combined dataset as our base task and the API vulnerability category dataset as our target task for the transfer learning scenario. We found that transfer learning decreased the f1-scores and accuracy of the projects of all the models. However, there was a slight increase in the accuracy of the CNN as more false negatives were detected. In this scenario, transfer learning did not provide any significant increase in the f1-score and accuracy. The decrease in the performance when applying transfer learning could be due to the discrepancy between the base task datasets and the target task dataset, which may have slightly affected the performance. When applying SMOTE to the dataset, we found that the f1-scores and accuracies of all the data dropped significantly by an average of 80%. The cause for the decline could be due to the lack of proper cleansing for the irrelevant points in the boundary classes to increase the separation between the two classes.

In summary, for RQ3, we can conclude that the attention mechanism provided the most gains due to the highest recorded scores compared to the other neural network models. This can be attributed to higher scores recorded from the neural network models.

4.4 | RQ4: Which Machine Learning Strategy provides the best performance improvement when paired with the Draper VDISC dataset?

For RQ4, we study the effects of the machine learning strategies on the Draper VDISC dataset. We study the impact of the machine learning strategies on the f1-score and accuracy recorded from the base scores and models. The VDISC Draper dataset consists of vulnerable and non-vulnerable examples of the CWE119, CWE120, CWE469, CWE476, and CWEOTHER vulnerability types. Table 5 shows the results of this experiment.

For the implementation of the attention mechanism, we found the f1-score of all the models dropped significantly for all the models by an average of 10% for the BGRU and BLSTM. There was also a 1.5% drop on the f1-scores of the CNN models. However, there was a slight increase of approximately 1% in the accuracy for all the datasets, excluding the CWE476 vulnerability type dataset, which experienced a slight drop in its accuracy. A similar case in the slight increase of the accuracy was seen in the results of the BLSTM. The accuracy of all the vulnerability type datasets, excluding CWE120 and CWE469, had a slight increase of approximately 1%. However, the application of attention in the CNN only increased the accuracy of the CWE469 vulnerability type dataset by 0.2%. Upon analyzing the results, we can determine that the attention mechanism did not perform effectively because of the source code representation. Lexed code representation represents data like tokens with relations rather than focus context. This may have affected the performance metrics of the models when attention is used.

For the application of the ensemble model, we wanted to identify if the model had higher accuracies and f1-scores for the dataset compared to the highest base model scores. We found a slight increase in the f1-score of the CWE469 vulnerability type dataset by approximately 6% and an increase in the accuracy of the CWE120 vulnerability type dataset by approximately 1.1%. However, the other datasets suffered a slight decrease in their accuracy and f1-score. This might have resulted from the ensemble model with decreased weights which may have slightly decreased the scores.

In summary, For RQ4, we can conclude that the attention mechanism provided the most gains since several of the vulnerability type dataset accuracies increased slightly upon its application on various neural networks as compared to the results of the ensemble model.

4.5 | RQ5: Which Machine Learning Strategy provides the best performance improvement when paired with the Devign dataset?

To answer RQ5, we study the effects of the machine learning strategies on the f1-score and accuracy recorded from the base scores and models. The Devign dataset consists of vulnerable and non-vulnerable examples of the FFmpeg, QEMU, and a combination of both. In terms of the base model scores, we found that the accuracy and f1-scores were significantly high for the BLSTM and BGRU but were slightly lower for the CNN model. The results computed for the learning strategies with the Devign dataset can be found in Table 7.

For the application of the attention mechanism, we found that for the BGRU. The QEMU dataset had a slight increase of 2% for its f1-score. The accuracy of the combined dataset improved by about 0.1%. However, the remaining f1-scores and accuracies had a slight drop of approximately 1%. The BLSTM experienced an increase of about 0.1% in the accuracy of the QEMU and combined datasets and faced a slight drop in the f1-scores. The CNN also experienced a decrease in the f1-score of the FFmpeg dataset but experienced an increase of 0.1% of the f1-scores of the QEMU and combined datasets project. The combined dataset also had a slight increase in its accuracy. Upon observing the results of using the SeqSelf-Attention layer, we found that the results had decreased slightly for the f1-scores, and accuracies may be due to the interpretation of relevant source code aspects between the input elements. The same might be a similar issue upon using the attention mechanism with the Devign dataset, thus leading to lower score averages.

We found that the datasets had a significant decrease in their f1-scores and accuracies upon applying dimensionality reduction. The f1-scores reported show a value of “nan,” and for the QEMU and combined dataset on the BLSTM and CNN, no false positives were detected in the results. However, the FFmpeg project experienced a significant increase in the f1-score by approximately 11% for the BLSTM, BGRU, and CNN. However, the accuracy for the other datasets had a drop of 15% on average. As the dimensions of the data were decreased, we had expected the f1-scores of the projects to face a significant reduction. However, most of the f1-scores reported a substantial decline in the result, and the accuracy suffered significantly. The slight increase in the f1-score for the FFmpeg dataset could be attributed to an increase in the recall for all the deep

learning models. Thus, demonstrating better predictions.

For the imbalance handling scenario using SMOTE, we found that the accuracy of all the projects on the deep learning models excluding the FFmpeg project of the CNN experienced a slight increase in their accuracy by about 5% on average. The f1-scores of the FFmpeg project for all the models suffered a loss of 2% on average. There was a slight increase for the f1-scores of the QEMU and combined dataset of the CNN with a slight increase of 2%. In summary, what could be concluded was that having synthetic examples created to balance out the classes for vulnerability detection resulted in a positive increase in the majority of the accuracies. This could also mean that the synthetic examples created best represented the actual data, thus increasing the accuracy of the results.

For the application of the ensemble model, we wanted to identify if the model had higher accuracies and f1-scores for the dataset compared to the highest base model scores. Upon comparing the results of the ensemble model to the base model, we found that the f1-score of the FFmpeg project of the ensemble model had a 7.5% increase in the f1-score compared to the best base model scores for the same project. The f1-score and accuracy for the combined dataset had a slight increase of 3% and 0.2%, respectively. The ensembled model only performed well for the combined dataset since the dataset size was larger. Since the ensemble models consist of a combination of the BRGU, BLSM, and CNN models, the decrease in performance for the individual datasets might be attributed to the shortcomings of the CNN when handling smaller dataset sizes.

In summary, For RQ5, we can conclude that imbalance handling provided the most gains due to the amount of increase in the f1-scores and accuracies of the projects compared to the other learning strategies.

5 | THREATS TO VALIDITY

External validity: The datasets selected and used for the experiment threaten the external validity of our work. The datasets used were taken directly from publicly available sources, thus making the dataset complaint with that used in the papers that had used them. Due to the large size of some of the datasets, the dataset may have defects that cannot be observed directly. However, issues arise from whether the dataset may have been mislabelled on some default values. Therefore, quality evaluation of the datasets is necessary. A scan was conducted to filter out faulty samples and ensure that the samples used were valid to overcome such threats with the datasets. For example, we used Joern [24] to compute the graph features for the SeVCs and skipped any examples that yielded compilation errors.

Internal validity: Biases in the interpretation of the evaluation metrics could hinder internal effectiveness. Due to the nature of the datasets in vulnerability detection being highly imbalanced and the adjustment of hyperparameters, there could be biases in the detection results. Factors such as these can cause misleading experimental results. To mitigate this problem, we randomly shuffle the dataset and conduct repeats in the training iterations. This helps us ensure that the models can have a better convergence to stable values in their performance, with each round having significantly different training sets generated.

Conclusion validity: Another issue arises with the validity of the conclusions drawn upon and the calculation metrics used. We used the method to determine which machine learning strategy fared well for a project by evaluating the amount of positive influence the learning strategy had on the set of projects representing each code representation. We used this approach as we felt that it could quickly gauge the amount of influence a learning strategy had on a particular code representation. For example, we were evaluating which learning strategy had the most impact on the CCRs. The assessment was mainly done using the f1-score and accuracy as inspired by the Devign paper. Another validity concern that may be brought about is the comparison of multiple code representations with each other. Since the code representations were of different datasets, we mitigated this concern by having our evaluations focus on each specific code representation.

6 | RELATED WORK

In deep learning for vulnerability detection, several works have utilized deep learning algorithms and source code representations. Lin et al. [46] obtained ASTs from three open sources GitHub projects, namely LibTIFF, LibPNG, and FFmpeg. Their process involved blurring out the project-specific name and generating the vector data to make their representations uniform. Code metrics were used to evaluate their deep learning models. Their work was further extended in [48], where the datasets were increased by adding data from 3 more projects, namely Pidgin, VLC, and Asterisk, in this extension of the project. Word2Vec embeddings were used as an addition to the project. The scenarios they tested covered,

evaluating whether transfer learned representations could perform better than Code Metrics and Random Selection.

Li et al.[18] proposed code gadgets as a code block-based feature representation through a model known as Vulnerability Deep Pecker (VulDeePecker) using a BLSTM and evaluated their model on four projects, namely Xen, Seamonkey, and Libav. The results showed that VulDeePecker was able to detect four vulnerabilities patched up by the developers of the projects.

Zou et al.[50] presented μVulDeePecker, an extension to the VulDeePecker paper, designed a new form of code representation called code attention, inspired by the notion of regions. Their model design had a high capability of pinning down the exact type of vulnerability found within a code gadget helps several analysts recognize vulnerabilities. Results showed that using their model and other systems such as VulDeePecker+ can capture more information for multiclass vulnerability detection. Ban et al.[23] used AST and code gadget representations in their performance evaluation experiment. The data used was collected from the projects FFmpeg, LibPNG, and, Asterisk for the ASTs. The code gadgets were under CWE119 and CWE399 provided by Li et al.[18]. Features are extracted from the feature vectors using a BLSTM network for classification on six machine learning classifiers. Besides, it was discovered that ML-based techniques. suffer poor performance on both imbalance handling scenario and cross-project problem.

Zhou et al.[8] proposed a graph neural network-based model called Devign. Their model was designed by incorporating the novel Conv model to extract useful information from rich node representations. Their model utilized a composite code representation, which is a combination of multiple graph-based source code representations. The results from the extensive evaluation on the datasets demonstrated that Devign outperforms other models by an average of 8.68% F1 score and 10.51% accuracy; this is further enhanced by the Conv module with a 4.66% accuracy increase and 6.37% F1 score increase.

7 | CONCLUSION

In this paper, we study how machine learning strategies can influence vulnerability detection for source code. Our experiments concluded that (i) The attention mechanism had the most positive influence in improving the vulnerability detection performance of our base models. (ii) the imbalance handling scenario only provided a positive increase in the f1-score and accuracy for various projects for the CCRs. However, imbalance handling did not improve the f1-scores and accuracies of the other representations. (iii) Transfer learning in the cross-project scenario did not positively increase the deep learning models' performance metrics. (iv) the learning strategy with the worse performance for most projects was dimensionality reduction. However, there is still room for improvement for the experiment:

- We focused on using readily available source code representations for most of our study. In the future, we shall emphasize creating different representations from multiple code sources to illustrate the models' effectiveness better.
- From the experiment, we limited it to using only a few of the learning strategies. In future work, this can be expanded to accommodate a wider variety of machine learning strategies. In addition, it would be essential to evaluate the effectiveness of combined representations in future works due to the coverage provided through capturing more information from the source code.
- Future works will investigate multiple programming language sources to provide a more holistic view of source code vulnerability detection.

8 | ACKNOWLEDGMENTS

This research was partially supported by the Key Laboratory of Advanced Perception and Intelligent Control of High-end Equipment, Ministry of Education (GDSC202006) and the 2021 Key R&D Program in Shaanxi Province (2021GY-041).

References

1. Shin Y, Meneely A, Williams L, Osborne JA. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering* 2011; 37(6): 772–787. doi: 10.1109/TSE.2010.81

2. Bilgin Z, Ersoy MA, Soykan EU, Tomur E, Comak P, Karacay L. Vulnerability Prediction from Source Code Using Machine Learning. *IEEE Access* 2020; 8: 150672–150684. doi: 10.1109/ACCESS.2020.3016774

3. Delaitre A, Stivalet B, Black P, Okun V, Cohen T, Ribeiro A. SATE V Report: Ten Years of Static Analysis Tool Expositions. https://www.nist.gov/publications/sate-v-report-ten-years-static-analysis-tool-expositions; 2018

4. Dowd M, McDonald J, Schuh J. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional . 2006.

5. Jimenez M. *Evaluating Vulnerability Prediction Models*. PhD thesis. University of Luxembourg, Luxembourg; 2018.

6. CVE - Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/; . Accessed: 2021-01-30.

7. Shen Z, Chen S. A Survey of Automatic Software Vulnerability Detection, Program Repair, and Defect Prediction Techniques. *Security and Communication Networks* 2020; 2020: 8858010. doi: 10.1155/2020/8858010

8. Zhou Y, Liu S, Siow J, Du X, Liu Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. https://arxiv.org/abs/1909.03496; 2019.

9. Ransomware WannaCry: All you need to know | Kaspersky.. https://www.kaspersky.com/resource-center/threats/ransomware-wannacry; . Accessed: 2021-01-30.

10. Ghaffarian SM, Shahriari HR. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys* 2017; 50(4). doi: 10.1145/3092566

11. CheckMarx Software Official Website. https://www.checkmarx.com; . Accessed: 2021-01-30.

12. Cadar C, Dunbar D, Engler D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: . 8. UNISEX. ; 2008: 209-224.

13. Pewny J, Schuster F, Bernhard L, Holz T, Rossow C. Leveraging Semantic Signatures for Bug Search in Binary Programs. In: ACSAC '14. Association for Computing Machinery. Association for Computing Machinery; 2014; New York, NY, USA: 406â“415

14. Henzinger TA, Jhala R, Majumdar R, Sutre G. Software verification with BLAST. In: Association for Computing Machinery. ; 2003: 235–239

15. Wu T, Wen S, Xiang Y, Zhou W. Twitter spam detection: Survey of new approaches and comparative study. *Computers and Security* 2018; 76: 265–284. doi: 10.1016/j.cose.2017.11.013

16. Sun N, Zhang J, Rimba P, Gao S, Zhang LY, Xiang Y. Data-Driven Cybersecurity Incident Prediction: A Survey. *IEEE Communications Surveys and Tutorials* 2019; 21(2): 1744–1772. doi: 10.1109/COMST.2018.2885561

17. Faghani MR, Nugyen UT. Modeling the Propagation of Trojan Malware in Online Social Networks. 2017; 15(June): 1–18.

18. Li Z, Zou D, Xu S, et al. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In: ; 2018: 1–15

19. Lin G, Wen S, Han QL, Zhang J, Xiang Y. Software Vulnerability Detection Using Deep Neural Networks: A Survey. *Proceedings of the IEEE* 2020; 108(10): 1825-1848. doi: 10.1109/JPROC.2020.2993293

20. Sabottke C, Suciu O, Dumitras T. Vulnerability Disclosure in the Age of Social Media: Exploiting Twitter for Predicting Real-World Exploits. In: USENIX Association. USENIX Association; 2015; Washington, D.C.: 1041–1056.

21. Scandariato R, Walden J, Hovsepyan A, Joosen W. Components via Text Mining. *IEEE Transactions on Software Engineering* 2014; 40(10): 993–1006.

22. Pang Y, Xue X, Namin AS. Predicting vulnerable software components through N-gram analysis and statistical feature selection. In: IEEE. IEEE; 2016: 543–548

23. Ban X, Liu S, Chen C, Chua C. A performance evaluation of deep-learnt features for software vulnerability detection. *Concurrency and Computation Practice and Experience* 2018; 31(19): 1–10. doi: 10.1002/cpe.5103

24. Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. In: IEEE. ; 2014: 590–604

25. Sorzano COS, Vargas J, Montano AP. A survey of dimensionality reduction techniques. https://arxiv.org/abs/1403.2877; 2014.

26. Sehgal S, Singh H, Agarwal M, Bhasker V, Shantanu . Data analysis using principal component analysis. In: No. 2. IEEE. IEEE; 2014: 45–48

27. Engler D, Chen DY, Hallem S, Chou A, Chelf B. Bugs as deviant behavior. *ACM SIGOPS Operating Systems Review* 2001; 35(5): 57–72. doi: 10.1145/502059.502041

28. Li Z, Zhou Y. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. *SIGSOFT Softw. Eng. Notes* 2005; 30(5): 306â“315. doi: 10.1145/1095430.1081755

29. Neuhaus S, Zimmermann T, Holler C, Zeller A. Predicting Vulnerable Software Components. In: CCS '07. Association for Computing Machinery. Association for Computing Machinery; 2007; New York, NY, USA: 529â“540

30. Jamali I, Bazmara M, Jafari S. Feature Selection in Imbalance data sets. 2012; 9(3): 5.

31. Zheng Z, Cai Y, Li Y. Oversampling method for imbalanced classification. *Computing and Informatics* 2015; 34(5): 1017–1037.

32. Amin A, Anwar S, Adnan A, et al. Comparing Oversampling Techniques to Handle the Class Imbalance Problem: A Customer Churn Prediction Case Study. *IEEE Access* 2016; 4(Ml): 7940–7957. doi: 10.1109/ACCESS.2016.2619719

33. Bahdanau D, Cho K, Bengio Y. Neural Machine Translation by Jointly Learning to Align and Translate. https://arxiv.org/abs/1409.0473; 2016.

34. Larochelle H, Hinton GE. Learning to combine foveal glimpses with a third-order Boltzmann machine. In: Lafferty J, Williams C, Shawe-Taylor J, Zemel R, Culotta A., eds. *Advances in Neural Information Processing Systems*. 23. NIPS. Curran Associates, Inc.; 2010.

35. Mnih V, Heess N, Graves A. Recurrent Models of Visual Attention. : 9.

36. Keras Self-Attention,. https://pypi.org/project/keras-self-attention/; . Accessed: 2021-01-30.

37. Raffel C, Ellis DPW. Feed-Forward Networks with Attention Can Solve Some Long-Term Memory Problems. 2015: 1–6.

38. Guidotti R, Monreale A, Ruggieri S, Turini F, Giannotti F, Pedreschi D. A Survey of Methods for Explaining Black Box Models. *ACM Comput. Surv.* 2018; 51(5). doi: 10.1145/3236009

39. Tan C, Sun F, Kong T, Zhang W, Yang C, Liu C. A survey on deep transfer learning. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2018; 11141 LNCS: 270–279. doi: 10.1007/978-3-030-01424-7_27

40. Ju C, Bibaut A, Laan v. dM. The relative performance of ensemble methods with deep convolutional neural networks for image classification. *Journal of Applied Statistics* 2018; 45(15): 2800–2818. doi: 10.1080/02664763.2018.1441383

41. Sorzano COS, Vargas J, Pascual A. A survey of dimensionality reduction techniques. 2014: 35.

42. Liu K, Kim D, Bissyandé TF, Yoo S, Traon YL. Mining Fix Patterns for FindBugs Violations. *arXiv:1712.03201 [cs]* 2018. arXiv: 1712.03201.

43. Russell R, Kim L, Hamilton L, et al. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In: IEEE; 2018: 757–762

44. Hovsepyan A, Scandariato R, Joosen W, Walden J. Software vulnerability prediction using text analysis techniques. In: Association for Computational Linguistics. ACM Press; 2012; Lund, Sweden: 7

45. Dam HK, Tran T, Pham T, Ng SW, Grundy J, Ghose A. Automatic feature learning for vulnerability prediction. 2017.

46. Lin G, Zhang J, Luo W, Pan L, Xiang Y. Poster: Vulnerability discovery with function representation learning from unlabeled projects. In: ; 2017: 2539–2541

47. Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *arXiv:1807.06756 [cs, stat]* 2018. arXiv: 1807.06756.

48. Lin G, Zhang J, Luo W, et al. Cross-Project Transfer Representation Learning for Vulnerable Function Discovery. *IEEE Transactions on Industrial Informatics* 2018; 14(7): 3289–3297. doi: 10.1109/TII.2018.2821768

49. Kim Y. Convolutional Neural Networks for Sentence Classification. In: Association for Computational Linguistics. Association for Computational Linguistics; 2014; Doha, Qatar: 1746–1751

50. Zou D, Wang S, Xu S, Li Z, Jin H. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *arXiv e-prints* 2020: arXiv:2001.02334.

How to cite this article: A.O.A. Semasaba, W. Zheng, X. Wu, S.A. Agyemang, T. Liuand Y. Ge (2020), An Empirical Evaluation of Deep Learning-Based Source Code Vulnerability Detection: Representation vs. Model, *Q.J.R. Meteorol. Soc.*, *2021;00:1–6*.