Representation vs. Model: What Matters Most for Source Code Vulnerability Detection

Wei Zheng
School of Software
Northwestern Polytechnical University
Xi'An, China
wzheng@nwpu.edu.cn

Abubakar Omari Abdallah Semasaba
School of Computer Science
Northwestern Polytechnical University
Xi'An, China
abu_sem@mail.nwpu.edu.cn

Xiaoxue Wu
School of Cyberspace Security
Northwestern Polytechnical University
Xi'An, China
wuxiaoxue00@gmail.com

Samuel Akwasi Agyemang
School of Software
Northwestern Polytechnical University
Xi'An, China
khermz2012@gmail.com

Tao Liu
School of Computer and Information
Anhui Polytechnic University
Wuhu, Anhui
liutiao@ahpu.edu.cn

Yuan Ge
School of Electrical Engineering
Anhui Polytechnic University
Wuhu, Anhui
geyuan@ahpu.edu.cn

***Abstract*— Vulnerabilities in the source code of 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 of complexity and volume. Deep learning has gained popularity throughout the years as a means of addressing such issues. In this paper, we propose an evaluation of vulnerability detection performance on source code representations and evaluate 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 when used in conjunction with source code representations for software vulnerability detection.**

Keywords—security, software vulnerability detection, deep learning

I. 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 operations of the system by users or developers, while defects are easier 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 and exploiting vulnerable systems to demand ransoms [6]. In today's environment, where the interaction 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 [8], the number of discovered vulnerabilities was less than 4600, while the vulnerabilities covered was almost 20000 [6, 7].

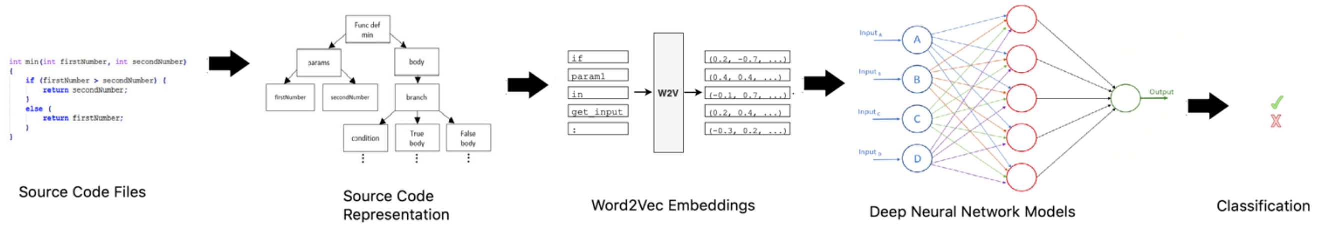
Our Contributions: For this paper, we conduct a performance evaluation for source code vulnerability detection on deep learnt features and evaluate how machine learning strategies can influence the model's performance. Our evaluation utilizes five source code representations, three deep learning models, 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 [7].

Second, we implement three Deep Neural Networks (DNNs), namely 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 [22] 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 is designed to include an embedding layer, a Bidirectional BGRU/LSTM Layer, GlobalMaxPooling1D, Dropouts, and Dense layers. (ii) The layers used for the CNNs consists 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) models 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 II is the brief introduction to deep learning-based vulnerability detection and discusses the ML strategies used. Section III discusses our experimental results, and Section IV presents a conclusion for this study.



```
int min(int firstNumber, int secondNumber)
{
    if (firstNumber > secondNumber) {
        return secondNumber;
    }
    else {
        return firstNumber;
    }
}
```

Fig. 1. Framework for deep feature learning model

II. DEEP-LEARNING BASED VULNERABILITY DETECTION

A. *Framework of software vulnerability analysis*

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 Fig 1. For the implementation of the ML strategies, the base models' structures are modified to accommodate them.

B. *Machine Learning strategies*

Imbalance Handling: In vulnerability detection, there arises a situation where the class distribution of the dataset has a skew problem. The class imbalance problem is a common problem that occurs in data mining [15]. In oversampling methods, minority class instances are increased in order to equate the classes. Common problems associated with oversampling methods is that they add new instances to the dataset, which results in overfitting classifiers. [16]. Amin et al. [17] used an oversampling method known as Synthetic Minority Oversampling Technique (SMOTE), which creates "synthetic" examples rather than replacing the minority class instances. SMOTE is commonly used as it performs better than both random and simple oversampling methods. For our experiment, we decided to use SMOTE as our imbalance handling learning strategy on the datasets before using them as inputs for our DNNs.

Attention Mechanism: The attention mechanism was first introduced for Natural Language Processing (NLP) tasks by Bahdanau et al. [18]. However, a similar idea called glimpses was proposed for computer vision by Larochelle and Hinton [19], 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. [20], 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 [10], and for the non-sequential neural network models, we use a custom attention layer inspired by Raffel et al. [24].

Transfer Learning: The idea behind transfer learning developed based on the phenomenon that deep neural networks trained on raw images learn similar features [23]. 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. We decided that cross-project transfer learning shall be applied to the POSTER dataset projects and the SySeVR dataset for our experiment. ALL_AST and ALL_SeVC projects are the base task, respectively. The target tasks are VLC and API projects, respectively.

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 different 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 [9]. For our experiment, the ensembled models consisted of a hybrid of CNN, BLSTM, and BGRU with tuned hyperparameters.

C. *Preprocessing the data*

Abstract Syntax Trees (AST): 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. [10], 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. Using the NVD description, we download the corresponding version of the project, locate each vulnerable function in the source code, and label it accordingly. For creating vectors of an equal length, the padding length of 650 is chosen. Vectors longer than 650 are truncated, and short vectors are padded at the end with zeros.

TABLE I. BASE MODEL RESULTS

Code Representation	Project	BGRU		BLSTM		CNN	
		F1 Score	*Accuracy*	*F1 Score*	*Accuracy*	*F1 Score*	*Accuracy*
AST	FFmpeg	nan	0.957831	nan	0.957831	nan	0.957831
	LibPNG	nan	0.941606	nan	0.941606	nan	0.941606
	LibTIFF	nan	0.872093	nan	0.872093	nan	0.872093
	ALL_AST	0.111111	0.956492	0.111111	0.956492	nan	0.954453
CG	CWE119	0.857515	0.931329	0.864109	0.932712	0.805563	0.910325
	CWE399	0.948299	0.965273	0.952447	0.968700	0.935236	0.957505
	ALL_CG	0.906806	0.945165	0.901983	0.944273	0.844492	0.918802
CCR	FFmpeg	0.551652	0.555783	0.594876	0.562947	0.580871	0.571647
	Qemu	0.541521	0.625641	0.589970	0.603988	0.555555	0.614814
	ALL_CCR	0.598614	0.596998	0.593847	0.596449	0.551831	0.579062
SeVCs	API	0.903213	0.961648	0.898929	0.960406	0.804754	0.922211
	ARRAY USE	0.901951	0.951219	0.904627	0.948022	0.840118	0.916765
	POINTER USE	0.914036	0.983484	0.915140	0.983964	0.732578	0.952269
	INTEGER OVERFLOW	0.900913	0.968178	0.908440	0.971112	0.837725	0.947190
	ALL_SeVC (Excluding API)	0.917199	0.978869	0.912686	0.977953	0.793167	0.944237
LCR	CWE119	0.370121	0.960243	0.327704	0.962212	0.467783	0.972332
	CWE120	0.420196	0.922801	0.411341	0.922965	0.480384	0.939839
	CWE469	0.029850	0.997960	0	0.998023	0.043321	0.997921
	CWE476	0.32	0.989598	0.255267	0.990296	0.349585	0.990162
	CWEOTHER	0.336176	0.947488	0.298455	0.962236	0.387540	0.960204

Code Gadgets (CG): To extract the CGs, library/API function calls are grouped into forward library/API function calls and backward library/API function calls. Forward library function calls receive one or more inputs directly from files or sockets. Backward library function calls; on the other hand, 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, which consists of 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. For data from NVD, the code gadgets are automatically labeled as 'vulnerable' using "1" if it has a statement deleted or modified or patched, then 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 that corresponds to a gadget is set to the value κ to ensure the vectors created are equal. When a vector is shorter than κ, backward slices of the code gadget are generated. The value of κ for our study is set to 50.

Semantics-based Vulnerability Candidates (SeVC (SeVC): In terms of the SeVCs used in this project, we follow the procedure by Li et al. [12]. 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, and so on. 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 CheckMarx 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 that is representative of the four primary vulnerability characteristics is used.

The next step is to convert SyVCs to SeVCs. SeVCs contain various statements that are semantically related to the SyVC based on control dependency, data dependency, or both. Joern does the extraction of PDGs for each SyVC in order to use the program slicing technique for the identification of 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 generate 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 transforming statements that belong to different functions into SeVCs.

SeVCs from the NVD is 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

TABLE II. TRANSFER LEARNING RESULTS

Code Representation	Project	BGRU		BLSTM		CNN	
		F1 Score	*Accuracy*	*F1 Score*	*Accuracy*	*F1 Score*	*Accuracy*
AST	VLC	nan	0.990573	nan	0.990573	nan	0.990573
	VLC_Transfer	nan	0.990573	nan	0.990573	nan	0.990573
SeVC	API	0.903213	0.961649	0.89893	0.960407	0.804754	0.922211
	API_Transfer	0.011364	0.842925	nan	0.842699	0.755906	0.923042

TABLE III. ATTENTION MECHANISM RESULTS

Code Representation	Project	BGRU		BLSTM		CNN	
		F1 Score	*Accuracy*	*F1 Score*	*Accuracy*	*F1 Score*	*Accuracy*
AST	FFmpeg	nan	0.957831	nan	0.957831	nan	0.957831
	LibPNG	nan	0.941606	nan	0.941606	nan	0.941606
	LibTIFF	0.370370	0.901162	0.370370	0.901162	nan	0.872093
	ALL_AST	0.111111	0.956492	0.111111	0.956492	nan	0.954452
CG	CWE119	0.860578	0.931455	0.865686	0.933341	0.774604	0.897748
	CWE399	0.949164	0.967329	0.947517	0.966186	0.923456	0.950422
	ALL_CG	0.907344	0.949140	0.904982	0.946787	0.792429	0.893251
CCR	FFmpeg	0.539148	0.551177	0.581485	0.555783	0.417230	0.515353
	Qemu	0.563049	0.617948	0.551376	0.619373	0.557263	0.598005
	ALL_CCR	0.583254	0.598279	0.560897	0.598828	0.565913	0.588396
SeVCs	API	0.895023	0.958077	0.900823	0.960717	0.800149	0.916931
	ARRAY USE	0.898641	0.946128	0.892551	0.942102	0.785247	0.891072
	POINTER USE	0.914045	0.983895	0.911221	0.983124	0.753299	0.952286
	INTEGER OVERFLOW	0.910384	0.971563	0.918595	0.975400	0.763125	0.912435
	ALL_SeVC (Excluding API)	0.919884	0.979408	0.918301	0.979095	0.767492	0.938038
LCR	CWE119	0.190964	0.974010	0.218846	0.972167	0.258277	0.962918
	CWE120	0.409495	0.928778	0.410954	0.920863	0.412541	0.922542
	CWE469	0	0.998023	0	0.998023	0	0.998015
	CWE476	0.175155	0.989582	0.080495	0.990680	0.244386	0.988648
	CWEOTHER	0.247691	0.96102	0.206761	0.965028	0.336417	0.952744

TABLE IV. IMBALANCE HANDLING RESULTS

Code Representation	Project	BGRU		BLSTM		CNN	
		F1 Score	*Accuracy*	*F1 Score*	*Accuracy*	*F1 Score*	*Accuracy*
AST	FFmpeg	nan	0.957831	nan	0.957831	nan	0.957831
	LibPNG	nan	0.941606	nan	0.941606	nan	0.941606
	LibTIFF	nan	0.872093	nan	0.872093	nan	0.872093
	ALL_AST	nan	0.954453	nan	0.954453	nan	0.954453
CG	CWE119	0.445399	0.350396	0.51592	0.510502	0.41422	0.262231
	CWE399	0.691482	0.704592	0.784143	0.818369	0.569523	0.498515
	ALL_CG	0.489017	0.396171	0.519015	0.464471	0.448502	0.289422
CCR	FFmpeg	0.542916	0.547595	0.586393	0.567554	0.571283	0.570624
	Qemu	0.597174	0.634473	0.529116	0.626781	0.559765	0.615954
	ALL_CCR	0.54958	0.60761	0.545304	0.598646	0.569324	0.582723
SeVCs	API	0.346441	0.212949	0.346143	0.212794	0.345195	0.208601
	ARRAY USE	0.412594	0.260005	0.421953	0.287946	0.412555	0.259886
	POINTER USE	0.177193	0.097209	0.177196	0.097226	0.177193	0.097209
	INTEGER OVERFLOW	0.280539	0.193184	0.281952	0.199954	0.271840	0.157301
	ALL_SeVC (Excluding API)	0.23465	0.13292	0.234649	0.132919	0.234649	0.132919

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".

For the creation of 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): The lexer designed by Russell et al. [13] 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

TABLE V. ENSEMBLED MODEL RESULTS

Code Representation	Project	Ensembled Network	
		F1 Score	*Accuracy*
AST	FFmpeg	nan	0.957831
	LibPNG	nan	0.941605
	LibTIFF	nan	0.872093
	ALL_AST	nan	0.954452
CG	CWE119	0.868201	0.932335
	CWE399	0.949705	0.966872
	ALL_CG	0.903207	0.944922
CCR	FFmpeg	0.646530	0.556806
	Qemu	0.557905	0.624786
	ALL_CCR	0.619958	0.599926
SeVCs	API	0.904948	0.962425
	ARRAY USE	0.905483	0.947549
	POINTER USE	0.904358	0.981993
	INTEGER OVERFLOW	0.915229	0.973369
	ALL_SeVC (Excluding API)	0.914384	0.978154
LCR	CWE119	0.435873	0.971429
	CWE120	0.475586	0.942875
	CWE469	0	0.998023
	CWE476	0.404643	0.988735
	CWEOTHER	0.374496	0.958596

TABLE VI. DRAPER VDISC DATASET

Draper VDISC dataset description	
CWE ID	*Vulnerability Description*
CWE119	Buffer Overflow
CWE120	Improper Restriction of Operations within the Bounds of a Memory Buffer
CWE469	NULL Pointer Dereference
CWE476	Use of Pointer Subtraction to Determine Size
CWEOTHER	Improper Input Validation, Use of Uninitialized Variable, Buffer Access with Incorrect Length Value, etc.

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, uint32_t, 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 are commonly used in, thus building the desired invariance. For our experiment, we use the Draper VDISC dataset, which consists of the projects shown in Table VI.

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. For our embedding layer, we use the embedding dimension size of 13 with five outputs based on the dataset's structure. Each of the outputs corresponds to the binary figures of “1” for being vulnerable and “0” for being not vulnerable.

Composite Code Representation (CCR): The idea behind composite code representation stems from the fact that program representations in program analysis manifest more in-depth semantic knowledge behind the textual code. Classical concepts such as ASTs, CFGs, and Data Flow Graphs (DFGs) capture the syntactic and semantic relationship among different tokens of source code [21]. The CCRs used by [7] consists of a joint structure that integrates ASTs, CFGs, Dataflow Graphs (DFGs), and Natural Code Sequence (NCS) into a joint structure.

For the datasets that were made available by the authors [7], first, we create a new data frame from the files and extract the columns “func” which represents the function and “target” which represents 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. The Devign dataset, which consists of the projects FFmpeg and Qemu, is used for our CCRs.

III. EXPERIMENTAL RESULTS AND ANALYSIS

A. *Experiment settings and metrics*

For this experiment, we decided that the evaluation metrics shall be the F1-score (F1) and accuracy (ACC) similar to that of the work of Zhou et al. [7]. Our experiment is run using 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. We considered three sizes for the training batch 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. Base model results are recorded in Table I.

B. *RQ1: Which source code representation has the best performance gain after the implementation of the attention mechanism?*

Regarding RQ1, for the POSTER dataset, we found that the base model accuracy and F1-score of the BGRU and BLSTM for the LibTIFF project improved by 0.370370 and 0.029069, respectively. For the remaining projects, there was no influence on the scores.

For the VulDeePecker dataset, the results of the BGRU reported an increase in F1-score for the CWE119, CWE399, and ALL_ASTs projects. The values recorded were 0.003063, 0.000865 and 0.000537, respectively. The accuracy for the same projects improved by 0.000125, 0.002056, and 0.003974, respectively. For the BLSTM, the F1-score for the CWE119 and ALL_ASTs improved by 0.001577 and 0.001824. The accuracy for the same projects improved by 0.000628 and 0.002514.

For the Devign dataset, the results for the BGRU showed an increase of 0.021528 for the F1-score of the Qemu project. In comparison, the ALL_CCRs project reported an increase of 0.00128 for accuracy. The BLSTM recorded an accuracy increase for the Qemu and ALL_CCRs with an improvement of 0.015384 and 0.002379, respectively. Regarding the CNN, the F1-score for the Qemu and ALL_CCRs increased by 0.001708 and 0.014082, respectively.

Concerning the SySeVR dataset, the F1- score and accuracy for the POINTER USE, INTEGER OVERFLOW, and ALL_SeVC projects for the BGRU increased. The reported increase for the F1-score of the projects was by 0.000009, 0.009470, and 0.002685. Simultaneously, the accuracy was reported to have increased by 0.000411, 0.003385, and 0.000539. For the BLSTM, the F1-score for the API, INTEGER OVERFLOW, and ALL_SeVC projects increased by 0.001893, 0.010155, and 0.005614, respectively. The accuracy of the same projects increased by 0.000310, 0.004287, and 0.001141. The CNN reported an F1-score and accuracy increase for the POINTER USE project by 0.020721 and 0.000017, respectively.

For the Draper VDISC dataset, the BGRU reported an increase in the accuracy of the projects CWE119, CWE120, CWE469, and CWEOTHER with reported increases of 0.013767, 0.005977, 0.000062, and 0.013531, respectively. The BLSTM model's accuracy for the CWE119, CWE469, and CWEOTHER projects increased by 0.009954, 0.000384, and 0.002792, respectively. The CNN had an accuracy increase of 0.000094 for the CWE469 project. The results for the attention mechanism are recorded in Table III.

Hence, the code representation with the best performance gain after having the attention mechanism added to the models is the SeVCs represented by the SySeVR dataset. The reason for selecting SeVCs is that several of the projects reported gains in both evaluation metrics compared to the other representations.

C. *RQ2: Which source code representation has the best performance gain after the implementation of ensembled models?*

For RQ2, we measure how the ensembled models can influence our base model scores' vulnerability detection performance. To determine which ensembled 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 (i.e., the ensembled model score is higher than the CNN, BGRU, and BLSTM score). For the POSTER dataset, there were no reported changes in the metrics. For the VulDeePecker dataset, the F1 score for the ensembled model used for the CWE119 project increased by 0.004092.

For the SySeVR dataset, the ensembled model's F1-score and accuracy increased for the API project by 0.001735 and 0.000776, respectively. An increase was also reported for the INTERGER OVERFLOW project with the values 0.006789 and 0.002256 for the metrics above. The ARRAY USE project had an increase of 0.000855 for its F1-score.

For the Devign dataset, the F1-score and accuracy for the Qemu project increased by 0.021343 and 0.020863, respectively. The FFmpeg project reported a slight increase in its F1-score, which increased by 0.051653. For the Draper VDISC dataset, the CWE120 project had an increase of 0.003035 for its accuracy. There is a slight increase to the F1-score of the CWE476 project, with an increase of 0.055058. The F1-score and accuracy for the CWE469 project remained the same as that of the BGRU project. The results for the ensembled models are recorded in Table V.

Hence, based on the influence of ensembled models on the F1 score and accuracy, we can conclude that the source code representation that performs the best is the SeVCs represented by the SySeVR dataset.

D. *RQ3: Which source code representation has the best performance gain after the implementation of imbalanced handling?*

Concerning RQ3, the Devign dataset reported an increase in accuracy for the BLSTM and BGRU models for all the projects. There was also an increase in both the accuracy and F1-score of the Qemu and ALL_CCRs project for the CNN. The results for the ensembled models are recorded in Table IV.

Hence, based on the influence of imbalance on the F1 score and accuracy recorded, we can conclude that the source code representation that performs the best is the CCRs represented by the Devign dataset.

E. *RQ4: How does cross project transfer learning influence the performance of the Deep Learning models with the source code representations for vulnerability detection?*

With RQ4, we considered the POSTER dataset and the SySeVR dataset. Upon using transfer learning with the Poster dataset, we find that transfer learning made no difference to the target project's F1 score and accuracy. Besides, for the SySeVR dataset, we found that transfer learning for the target project with the CNN increases the accuracy by 0.000831. The results for transfer learning are recorded in Table II.

Hence, we can conclude that transfer learning for the cross-project scenario negatively influences the deep learning models' performance.

F. *RQ5: Which ML strategy has the best result for improving vulnerability detection performance?*

For RQ5, we study which machine learning strategy had the most positive influence on the projects when paired with the deep learning models. We evaluate how many projects from each representation increased the F1 score or accuracy to answer this question.

Hence, we concluded that the machine learning strategy that is the best for improving the vulnerability detection metrics is the attention mechanism.

IV. 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 CCRs had the best overall improvement when paired with deep learning models, and machine learning strategies (iii) Transfer learning in the cross-project scenario did not positively increase the deep learning models' performance metrics.

Acknowledgment

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 Innovation Foundation for Doctor Dissertation of Northwestern Polytechnical University (CX202067).

REFERENCES

[1] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity code churn and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772-787, 2011.

[2] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay, "Vulnerability Prediction From Source Code Using Machine Learning," *IEEE Access*, vol. 8, pp. 150672-150684, 2020.

[3] M. Delaitre, B. C. Stivalet, P. E. Black, V. Okun, T. S. Cohen, and A. Ribeiro, ''Sate V report: Ten years of static analysis tool expositions,'' NIST, Gaithersburg, MD, USA, Tech. Rep. SP-500-326, 2018.

[4] M. Dowd, J. McDonald, and J. Schuh, The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Boston, MA: Addison-Wesley, 2007.

[5] M. Jimenez, ''Evaluating vulnerability prediction models,'' Ph.D. dissertation, Dept. Sci., Technol. Commun., Univ. Luxembourg, Rue Mercier, Luxembourg, Oct. 2018.

[6] Z. Shen and S. Chen, "A Survey of Automatic Software Vulnerability Detection, Program Repair, and Defect Prediction Techniques," *Security and Communication Networks*, vol. 2020, p. 8858010, Sep. 2020, doi: 10.1155/2020/8858010.

[7] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, ''Devign: Effectivevulnerability identification by learning comprehensive program semanticsvia graph neural networks,'' in *Proc. Adv. Neural Inf. Process. Syst.*, New York, NY, USA: Curran Associates, 2019, pp. 10197–10207. [Online]. Available: http://papers.nips.cc/paper/9209-devign-effective-vulnerability-identification-by-learning-comprehensive-program-semantics-via-graph-neural-networks.pdf

[8] "CVE - Common Vulnerabilities and Exposures (CVE)." [Online]. Available: https://cve.mitre.org/

[9] C. Ju, A. Bibaut, and M. V. D. Laan, "The relative performance of ensemble methods with deep convolutional neural networks for image classification," *J. Appl. Stat.*, vol. 45, no. 15, pp. 2800–2818, 2018, doi: 10.1080/02664763.2018.1441383.

[10] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "Poster: Vulnerability discovery with function representation learning from unlabeled projects," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 2539–2541, doi: 10.1145/3133956.3138840.

[11] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. NDSS*, San Diego, CA, USA, 2018, pp. 1–15. = 11

[12] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," *arXiv:1807.06756 [cs, stat]*, Sep. 2018, Accessed: Dec. 16, 2020. [Online]. Available: http://arxiv.org/abs/1807.06756.

[13] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Orlando, FL, USA, Dec. 2018, pp. 757–762.

[14] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction," *arXiv:1708.02368 [cs]*, Aug. 2017, arXiv: 1708.02368. [Online]. Available: http://arxiv.org/abs/1708.02368

[15] I. Jamali, "Feature Selection in Imbalance data sets," *IJCSI: International Journal of Computer Science Issues*, vol. 9, no. 3, pp. 42-45, May 2012.

[16] Z. Zheng, Y. Cai, and Y. Li, "Oversampling method for imbalanced classification," *Computing and Informatics*, vol. 34, pp.1017-1037, 2015.

[17] A. Amin, S. Anwar, A. Adnan, M. Nawaz, N. Howard, J. Qadir, A. Hawalah, and A. Hussain, "Comparing Oversampling Techniques to Handle the Class Imbalance Problem: A Customer Churn Prediction Case Study," *IEEE Access*, vol. 4, pp. 7940-7957, 2016, doi: 10.1109/ACCESS.2016.2619719.

[18] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. ICLR*, 2015, pp. 1–15.

[19] H. Larochelle and G. E. Hinton, "Learning to combine foveal glimpses with a third-order Boltzmann machine," in *Proc. NIPS*, 2010, pp. 1243–1251.

[20] V. Mnih, N. Heess, and A. Graves, ''Recurrent models of visual attention,'' in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 2204–2212.

[21] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings - IEEE Symposium on Security and Privacy*, 2014, pp. 590–604, doi: 10.1109/SP.2014.44Y.

[22] Kim, "Convolutional Neural Networks for Sentence Classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar, 2014, pp. 1746–1751, doi: 10.3115/v1/D14-1181.

[23] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu, "A survey on deep transfer learning," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 11141 LNCS, pp. 270–279, 2018, doi: 10.1007/978-3-030-01424-7_27.

[24] C. Raffel and D. P. W. Ellis, "Feed-Forward Networks with Attention Can Solve Some Long-Term Memory Problems," *arXiv:1512.08756 [cs]*, Sep. 2016, Accessed: Jan. 12, 2021. [Online]. Available: http://arxiv.org/abs/1512.08756.