Predicting Source Code Vulnerabilities Using Deep Learning: A Fair Comparison on Real Data

Vincenzo Carletti[1],*, Pasquale Foggia[1],*, Alessia Saggese[1] and Mario Vento[1]

[1]Dept. of Computer Engineering, Electrical Engineering and Applied Mathematics, University of Salerno, Via Giovanni Paolo II, 132, 84084, Fisciano (SA), Italy

Abstract

In the context of software development, the detection of vulnerabilities within source code is a paramount concern, especially for programming languages like C and C++ that are widely used in mission-critical applications, operating systems and embedded software. Traditional approaches to detecting vulnerabilities in source code often struggle due to their reliance on hand-crafted rules and pattern matching, which can lead to high rates of false positives and require a considerable effort by human experts. Additionally, the evolving nature of software development practices and the increasing sophistication of cyber threats constantly challenge traditional systems, making them less and less useful over time. In this paper we explore the effectiveness of state-of-the-art deep learning methods in identifying vulnerabilities within C/C++ source code of real-world software projects. We have conducted a comprehensive analysis comparing basic deep learning methods used for text processing against more advanced architectures, including Transformers and Graph Neural Networks (GNNs), aiming to provide a reliable benchmark for evaluating vulnerability detection approaches. To this purpose we have prepared a large dataset, combining and normalizing data from several publicly available code datasets extracted from well-known open-source software projects, namely Big-Vul, DiverseVul, Devign and ReVeal. The results of the analysis provide insights about the complexity of the task at hand when faced in a realistic setup and suggest some challenges and promising research directions to use the most recent deep learning models.

Keywords

Vulnerability Detection, Deep Learning, Graph Neural Networks,

1. Introduction

Software vulnerabilities are, together with misconfigurations, among the most relevant issues in cybersecurity due to the potential impact they can have in the security of systems and networks, leading to significant financial losses, reputation damage, and also threats to physical security when the system under attack is mission critical.

Statistics from the National Vulnerability Database (NVD) [1] indicate that in the period from 2021 to 2022, there were over $33,000$ publicly disclosed vulnerabilities; of these, over $7,000$ were rated as having High or Critical severity according to the Common Vulnerability Scoring System (CVSS).

The search for vulnerabilities is commonly performed by domain experts, often with the help of tools that perform a *static* or *dynamic analysis* of the program. Static analysis tools examine the source code (or, less often, the binary code) of the program without executing it, looking for dangerous code patterns [2, 3, 4, 5]. Dynamic analysis tools, instead, execute the program in a controlled environment, detecting dangerous behaviors. Both static and dynamic analysis tools are traditionally based on the use of hand-crafted rules, devised by domain experts, and may need a careful tuning to balance missed detections and false positives. Hybrid approaches, that combine static and dynamic analysis, are also available but they often suffer from the same drawbacks [6, 7].

In the current cybersecurity scenario, where malicious users incessantly devise new ways to exploit known or previously undiscovered vulnerabilities (the so-called *zero day vulnerabilities*), companies are strongly interested in the research and development of effective and accurate tools for software

ITASEC 2024: The Italian Conference on CyberSecurity

[*] Corresponding author.

vcarletti@unisa.it (V. Carletti); pfoggia@unisa.it (P. Foggia); asaggese@unisa.it (A. Saggese); mvento@unisa.it (M. Vento)
0000-0002-9130-5533 (V. Carletti); 0000-0002-7096-1902 (P. Foggia); 0000-0003-4687-7994 (A. Saggese); 00000-0002-2948-741X (M. Vento)

vulnerability detection. For instance, big software manufacturers periodically promote bug hunting campaigns where experts are invited, under reward, to search for software bugs and vulnerabilities. Additionally, with software growing in complexity, it is becoming more challenging to find, manage and patch bugs, making it difficult to keep the pace with the rapid rise of new vulnerabilities [8, 9].

While rule-based approaches are struggling to face with such a challenging context, machine learning methodologies have demonstrated to achieve remarkable performance [10]. Nevertheless, to accomplish this goal they still need the involvement of domain experts to design a suitable representation of the program in terms of measurable *features*, that can be fed as input to a machine learning algorithm. This process, commonly named *feature engineering*, is time-consuming and its effectiveness strongly relies on the knowledge of the expert on both the specific problem and the adopted machine learning method [11, 12, 13]. For this reason, the scientific community has recently moved the focus to deep learning approaches, capable of autonomously learning the best representation from the data. Such methods reduce the need for domain expert involvement, and limit it mainly to data preparation and validation phases, since their performance heavily depends on the quantity and quality of the data and on the adopted training strategy [14]. In general, deep learning approaches have demonstrated to be more effective and accurate as well as potentially capable to generalize on new unseen kinds of vulnerabilities [15, 16, 10, 17].

Among the deep learning models, several recent approaches are based on Graph Neural Networks (GNNs) [18, 19, 20, 21, 22, 23], i.e. neural networks designed to process data structured as graphs instead of vectors. The idea behind these methods lies on the fact that source code can be naturally represented as a graph, for instance using Abstract Syntax Trees (AST) or Control Flow Graphs (CFG), and that the adoption of graph-based representations allows to take into account both semantic and syntactic structures [24].

The current literature lacks standard benchmarks and datasets that are representative of the problem at hand, as highlighted in a recent work by Chakraborty et al. [20]. Specifically, many datasets are designed for educational or demonstrative purposes, so they are composed of artificially created vulnerable functions. On the other hand, those datasets that contain real code are often noisy, biased and inaccurate since the labelling is performed by using existing tools for static analysis without manually reviewing their output, so they lead to machine learning models that are biased and excessively prone to false positives. Furthermore, data are naturally unbalanced, with a large amount of negative samples (i.e. non vulnerable code), so causing the model to shift towards false negatives.

When the desired output is not limited to the presence or absence of a vulnerability, but includes the determination of the kind of vulnerability, all these issues are amplified due to both a strongly unbalanced distribution of the classes (some kinds of vulnerabilities are much rarer than others) and an inherent degree of ambiguity in the classification. These problems bring others with them, first of all the difficulty of having a fair and clear assessment and comparison of deep learning methods for the problem at hand. Therefore, the main purpose of this paper is to provide an analysis aimed at assessing if deep learning approaches can be effectively exploited to detect vulnerabilities in C/C++ source code extracted from real software projects. Previous analysis of some state-of-the-art deep learning methods have been proposed in [25, 20].

Differently from them, we have built a significantly larger set of data composed of 396,130 samples, by thoroughly revising and cleaning the data belonging to four public datasets of source code extracted from open-source projects: Big-Vul [26], DiverseVul [27], Devign [25] and ReVeal [20]. In our analysis, we have selected and compared basic deep learning methods with more complex neural network architectures such as Transformers and GNNs, using a demanding and realistic experimental setup. In particular, we evaluated these models in scenarios where the test set samples originated from software projects whose code was not in the training set.

This paper is organized as follows. In Section 2 we will provide an overview of state-of-the-art approaches and representations proposed for source code analysis and vulnerability detection tasks. In Section 3 we will present the experimental setup describing the dataset and introducing the methods considered in the analysis. In Section 4 we will present and discuss the results obtained. Finally, in Section 5 we will provide some conclusions and future directions.

2. Vulnerability Prediction Through Deep Learning

A *vulnerability* is an occurrence of a *weakness* in a piece of software, where a *weakness* is a mistake (usually in the implementation or in the design of the code) that may make a security threat possible. For example, accessing an array with an index outside of the allowed bounds is a weakness; the presence of this weakness inside a particular subroutine of a program is a vulnerability. In order to facilitate the communication among security practitioners, known weaknesses are listed in a community-maintained catalog, the Common Weakness Enumeration (CWE) where each weakness has a unique identifier and a detailed description. Also, the CVE Program (Common Vulnerabilities and Exposures) maintains a list of publicly disclosed vulnerabilities, that are made available for consultation through several services; among them, the National Vulnerability Database (NVD) [1].

The problem of detecting a vulnerability can be formulated in different ways on the base of the desired result: it can be considered as a binary classification task if we only need to know whether a piece of code contains a vulnerability or not; it becomes a multi-label classification if, in addition, we are interested in which particular weakness causes the vulnerability (for instance, reporting the corresponding CWE identifier). In this Section, we will introduce the most recent approaches and representations to address the task by considering both those derived from the Natural Language Processing (NLP) field and those relying on graph-based methods.

2.1. Natural Language Processing Approaches

Traditionally, code analysis is addressed as a *Natural Language Processing (NLP)* problem where code snippets are transformed in sequence of words, a.k.a. *tokens*. Each word is represented using a high-dimensional vector space through *word embedding* methods [28]; thus, a code snippet corresponds to a sequence of embedding vectors. In several methods, these sequences are then processed by a Recurrent Neural Network (RNN) to learn dependencies among the elements in the sequence and to encode them in an enriched vector representation; the latter is finally exploited by a neural classifier (for instance, a Multi-Layer Perceptron) to predict if the original code snippet is vulnerable, and possibly to assign a label representing the CWE identifier. In more recent methods, the first part of this process is performed replacing RNNs with Transformers [25], that are advanced deep neural architectures for sequence-processing, based on an attention mechanism [29], that allows the network to learn which are the most relevant parts of the input sequence for the determination of the output. μVulDeePecker, introduced by Zou et al. [30], is a notably and recent example of a method based on Long Short-Term Memory (LSTM), a modified version of an RNN. The LSTM is used to merge and relate global and local features extracted from the source code of a software. The latter is initially decomposed into code gadgets, i.e. semantically related statements, not necessarily consecutive, that satisfy data and control dependency relations, to capture global features. In addition to gadgets, the authors also introduce the concept of *code attention*, inspired by the concept of regions of interest in image processing, that aims to capture regions of code that can be relevant to detect a specific vulnerability, like API function call, control statements or statements referring to libraries. More recently, the authors of [17], have proposed VulBERTa, a transformer-based approach, designed to learn a deep representation of C/C++ code, capturing both syntax and semantic elements. The neural network is built upon RoBERTa, a language model proposed in [31] as an extension of the well-known BERT [25]. The proposed architecture is trained in two phases: pre-training, where the network learns a first representation from the unlabeled code examples, using RoBERTa; and fine-tuning, that refines this representation to perform the classification task jointly with a Multilayer Perceptron (MLP) or a Text Convolutional Neural Network (TextCNN); the two versions are named VulBERTa-MLP and VulBERTa-TextCNN, respectively.

2.2. Graph-Based Approaches

Although NLP approaches have been demonstrated to achieve remarkable performance in processing software programs as text, they can neglect syntactic structures and relations among different parts

```
int foo(int x) {
    int y;
    if (x > 0) {
        y = x * 2;
    } else {
        y = -x;
    }
    return y;
}
```

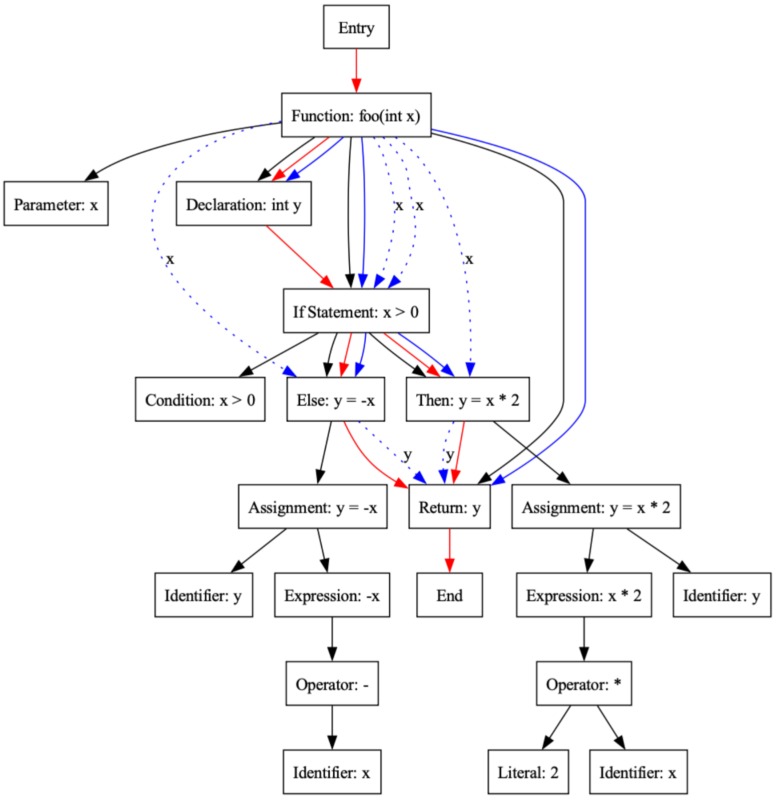

Figure 1: Example of CPG graph extracted from the source code on the left. The graph contains edges of AST in black, CFG in red and PDG in blue. The data dependencies are represented as dashed lines and while control dependencies with solid lines

of the source code. Differently, graph-based representations can be more effective in capturing both semantic and syntactic structures [24]. This is motivated by the fact that a software program can be naturally represented as a graph; actually, several graph-based representations can be used, depending on the level of abstraction on which we want to focus. Indeed, each entity in the source code, from single keywords and and individual statements to entire subroutines, can be represented as a node; and every kind of relationships between such entities can be represented as edges. How we represent the source code depends on the specific purpose and affects the way we intend entities and relationships in the graph, therefore, different graph-based representations of the source code have been proposed in the last decade for the purpose of automatic learning of code properties; among them the most commonly adopted are Abstract Syntax Trees (AST) [32], Control Flow Graphs (CFGs) [33], Program Dependency Graphs (PDGs) [34]. Since each of these representations has limits in its expressiveness, that can lead to an inaccurate characterization of the vulnerability in complex scenarios, in [7] the authors propose to merge all of them in the Code Property Graphs (CPGs) so as to provide a comprehensive graph to analyze the source code at multiple levels of abstraction simultaneously. In Figure 1 we show how a code snipped is represented by using a CPG graph.

Over the last decades, several methods have been proposed to use machine learning methods on graphs [35, 36, 37]. Nowadays, state-of-the-art approaches are based on GNNs [38], that are neural networks able to work directly on graph data, without the need to previously construct a feature vector summarizing the information contained in the graph. GNNs can be trained to produce a vector representation associated to each node, to each edge and/or to the entire graph (i.e. a *node*,*edge* or *graph embedding*), where the vectors encode both the semantic and the structural information contained in the graph (e.g. the neighbors of a node); these embeddings can then be used by another neural network to perform different tasks, such edge and node prediction or graph classification. The latter is the most common formalization for the vulnerability prediction problem.

The first method to face the task at hand using a GNN has been Devign [18]. The architecture of this GNN is composed of three layers: a code embedding layer that turns the source code into a graph, followed by a sequence of Graph RNN layers (a.k.a. Gated Graph Recurrent Layers), used to learn a new vector representation for the nodes, and a final convolutional layer that processes the matrix containing the node vectors produced by the previous layer.

Successively, Chakraborty et al. [20] have proposed ReVeal. It firstly transforms the source code into a CPG; the code snippets contained in the nodes of the CPGs are then embedded in vectors using word2vec [28], a neural network designed for word embedding. For each node, the embeddings of the corresponding tokens are concatenated, getting the initial node representation. Finally, the obtained graph is fed into a GNN to get a vector representation of the entire graph, that is used for the classification.

2.3. Hybrid Approaches

A further approach that can be found in the recent literature is to combine Transformers and GNNS. A first noteworthy method, namely VELET, has been proposed by Ding et al. [21], with the aim to take into account both local and global contexts of statements in C/C++ function source code. VELVET firstly builds a CPG and then extracts a vector representations for the nodes using word2vec similarly to ReVeal. Node embeddings are processed simultaneously by a GNN and a Transformer to update node representations and their output is independently processed by a classifier. The prediction are combined using a final stage named *Embedding and Ranking* that provides the label to the code.

In [22] Hin et al. propose LineVD, an approach is similar to VELVET. The overall architecture mostly differs in the final stage, indeed the outputs of the Transformer and the GNN are processed through a neural network that also provide the final prediction. LiveVD is based on CodeBERT [39] a Transformer trained to embed statements and functions as vectors, and a Graph Attention Network (GAT), a GNN that exploits the attentions mechanism during the aggregation of node's neighborhood to weight differently each neighbor on the base of its relevance.

3. Experimental Framework

As introduced in Section 1, the main purpose of this work is to evaluate if deep learning methods can be considered as an effective tool to predict source code vulnerabilities, indeed by analyzing the state-of-the-art there are significant lacks that do not allow us to infer a clear answer. In this section we describe how we have prepared the dataset and how we have used it to train and compare the selected methods.

3.1. Dataset Preparation

Since the most relevant issue is the data, we firstly focused our effort in preparing a dataset that can provide significant and comparable results, starting from those are publicly available. We excluded those datasets containing synthetic or semi-synthetic functions, as well as those labeled exclusively through automatic tools; the final choice has been to consider only samples belonging to Devign [18], Big-Vul [26], DiverseVul [27], and ReVeal [20]. We named this new dataset the *MIVIA Vulnerable Functions Source Code (MVFSC)*; in Table 1, we have reported a detail of its composition.

Devign is composed of functions from large-scale, real-world open-source projects, such as FFmpeg, QEMU, Linux Kernel, and Wireshark. Each function has been manually labeled by a team of three security experts over seven years to reduce false positives and negatives coming from automated static analysis. However, only a small subset of the Devign, containing samples of the Linux Kernel and the FFMpeg projects, is publicly available. Big-Vul has large and highly heterogeneous amount of samples extracted from 348 different software projects, including Chromium, Linux Kernel, and

Android. Differently to Devign that has only a binary label (vulnerable or not), Big-Vul also includes, for a subset of the samples, the CWE classification. Furthermore, since the vulnerabilities in Big-Vul are documented in the CVE database, their authenticity is effectively confirmed. DiverseVul has been built similarly to Big-Vul, containing mostly source code from real, large, and open-source projects by attaching CWE to vulnerable code. The labelling has been performed through the analysis of security issue websites and commits related to vulnerability fixes. According to the authors, DiverseVul includes high-quality commits from various open-source projects, making it an effective tool for training models capable of generalizing on unseen data. Finally, ReVeal has been prepared by extracting historical vulnerabilities of two large and relevant open-source projects, namely the Debian and Chromium projects. The dataset contains 22,734 samples among them former there are several type of weaknesses such as buffer overflows, format string issues, and integer overflows.

Table 1
Composition of the dataset selected to build the one used for the experiments. Finally, we reported the composition of the MVFSC

Dataset	Vulnerable Functions	Neutral Functions	Labelling
Devign	12,460	14,858	Binary
BigVul	11,823	253,096	Multi-class
DiverseVul	18,945	330,492	Multi-class
Reveal	2,240	20,494	Binary
MVFSC	31,985	364,145	Multi-class

The preparation of this new dataset required a meticulous data processing. Since some datasets may contains samples extracted from the same software projects, we have to firstly remote duplicated functions to avoid redundancies or inconsistent labelling. The detection of duplicated functions has been made by computing the hash for each of them, so functions with duplicate hashes has been made unique under the assumption that duplicates originated from the same commit or project. In case of duplications with different labels we have removed these samples for the dataset to avoid ambiguities. Successively, with the aim of preparing the data to be properly processed by work embedding neural networks, we have cleaned the functions by removing carriage returns, new line feeds, tabs and excess spaces. This operation has been done very carefully to keep the original syntax and semantics of the source code unchanged, while maintaining the structural and functional integrity of the code. To prevent the models to be biased by useless information we also removed the comments within the code; they may contains clarifications for humans as well as subjective, irrelevant, or potentially misleading information. Since we also have to extract graph-based representation from the source code using automatic code parsing tools the last step has been to remove the C/C++ functions that were not properly recognized by these tools. For the sake of clarity, the training set is composed of the data belonging to Devign, BigVul and DiverseVul, while ReVeal has been used as test set only.

At the end of the data preprocessing, the training set results in 396,130 samples, of which 364,145 belong to the neutral class and 31,985 belong to the vulnerable class. Among the vulnerable samples, 15,656 are labeled with a specific CWE. The test set consists of a total of 19,762 samples, of which 17,786 are labeled as neutral and 1,916 as vulnerable. It is worths to point out that this dataset is highly unbalanced, but being prepared from real software project it reasonably represent the real a-priori distribution of the data, a fundamental requirement to properly train a machine learning model.

Then a graph-based representation have been extracted from the source code samples using Joern [40], a well-known open-source framework for static code analysis designed to extract graphs from large software project. It processes the source code by decomposing it in an AST graph, where each node is a block of code, then the representation is extended with additional information, such as dependencies between variables and execution paths to get finally a CPG graph.

3.2. Deep Learning Approaches

As previously introduced, in the proposed analysis we have considered different kind of machine learning approaches from basic recurrent neural networks, like LSTM to more complex architectures such as Transformers and GNN. On the one hand we have taken into account basic methods to assess the complexity of the problem at hand, thus to understand if common baseline approaches are sufficiently effective; on the other hand, we aimed at understanding if attention mechanisms and structural approaches, like GNN, can provide benefits when dealing with such a challenging problem.

More specifically, we have used Word2Vec [28] (w2v) as source code embedding method for all the considered approaches since, all of them, required to turn the text into vectors in order to process the source code. Therefore, we started from a the pre-trained model of w2v and we have fine tuned it for the task at hand so as to get more significant vector representations of the token that are commonly in the C/C++ language.

Once we have a sequence of vectors corresponding to the sequence of tokens in the source code, we have build a neural network based on an LSTM and a fully connected network as output stage to predict if a given function is vulnerable. As an alternative, to this architecture, we have also used a Bidirectional LSTM (BiLSTM) to evaluate if taking into account the dependencies among the tokens in both the directions can help to obtain a more discriminative representation to be fed to the fully connected layer. Both the approaches have been trained end-to-end together with w2v.

Another baseline approach that can be easily adopted to the problem at hand is TextCNN, a convolutional neural network (CNN) proposed in [41, 42] for sentence classification. This approach, initially designed for sentiment analysis has been recently applied also to malware code classification in [43]. It uses w2v to build a vector representation of the words in a sentence then the obtained vectors are arranged in a matrix and processed by a CNN, similarly as it is performed for images, the a fully connected layer provides the label.

In addition to the previous models, we have analyzed how GNNs are capable to perform on the task at hand. To this purposed we have considered to use a state-of-the-art message passing GNN, namely GraphSage [42], to process the GPG graphs extracted from the source code. It worth to note that in the latter the node contains the source code as text and it cannot be processed by a GNN directly. Therefore, we used w2v to extract a vector representation for all the words in the code snippet related to a node, then the feature vectors have been summarized into a single vector adopting two approaches: an average pooling layer and a BiLSTM. The graph embedding produced by the GraphSage has been then processed by a pooling layer to obtain a vector representation for the whole GPG graph end finally fed to a fully connected stage to provide the classification.

Finally, we have fine-tuned VulBERTa [17], using the proposed dataset to have results comparable with the other approaches and evaluate if such a complex architecture can really provide benefits. In the analysis we have taken into account both VulBERTa-MLP and VulBERTa-TextCNN. In addition, inspired by the ReVeal architecture proposed in [20], we have also explored the possibility to use the VulBERTa Transformer as an embedding network to produce a CPG graph. The obtained graphs have been processed by a Gated Graph Neural Network (GGNN), a variant of GNNs, inspired by gating methods used to enhance the stability and learning capabilities of recurrent neural networks (RNNs), such as LSTMs. The used architecture has been designed with two GGNN layers separated by a TopK pooling layer. The produced graph embedding is then processed by a fully connected layer.

For all the neural networks, the output of the fully connected layer is passed to a soft-max function and we have used a *binary cross-entropy loss function.*

4. Results

In Table 4 we report the results of the experimental analysis considering different well-known metrics. In addition to accuracy (Eq. 1), defined as the number of correct predictions with respect to the total amount of samples processed, we have also reported *precision* (Eq. 2), *recall* (Eq. 3), and *F1-Score* (Eq. 4).

Table 2
Results of the experimental analysis on the test set.

Method	Precision	Recall	F1-Score	Accuracy
w2v + LSTM	0.16	0.53	0.25	0.69
w2v + BiLSTM	0.17	0.59	0.27	0.68
w2v + TextCNN	0.24	0.53	0.33	0.83
w2v + GraphSAGE	0.55	0.82	**0.65**	**0.93**
w2v + BiLSTM + GraphSAGE	0.27	**0.84**	0.40	0.76
VulBERTa-TextCNN	0.19	0.68	0.30	0.73
VulBERTa-MLP	0.57	0.67	0.55	0.70
VulBERTa-GGNN	**0.60**	0.75	0.59	0.72

The latter are relevant to problem at hand since the since the dataset is highly unbalanced, so using only the accuracy can lead to misleading interpretations of the actual performance.

For the sake of clarity, we have considered as a *true positives* (TPs) and *true negatives* (TNs) are vulnerable and non-vulnerable samples classified as such respectively, while *false positives* (FPs) are those non-vulnerable instances misclassified as vulnerable and *false negatives* (FNs) are vulnerable samples labelled as non-vulnerable. In the following equations we report how the performance metrics have been computed using the number of TPs, FPs, and FNs.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3}$$

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

The complexity of the task at hand when performed on real-world data is confirmed by the fact that all the baseline deep learning approaches, that are commonly adopted for text processing, have achieved a very low performance. Among these, the best one in terms of accuracy and F1-Score is TextCNN with the word embedding obtained through w2v. From the results, it emerges that using the VulBERTA Transformer instead of w2V does not provide any advantage to TextCNN, since there is drop on both the metrics. This ineffectiveness of VulBERTA with respect to a simpler model like w2v can be related to the limited amount of vulnerable samples. On the other hand, when using a simpler output neural network like the MLP, VulBERTA we have obtained the most balanced performance between precision and recall among all the considered non-structural approaches suggesting that the Transformer is able to learn a good representation of the code.

The main evidence from the results in Table 4 is that using a graph-based representation of the source code allows to significantly improve the performance on the task. In all the cases it is possible to note an increase of the recall, suggesting that the model is less prone to false negatives. In particular, in the case of the methods w2v + GraphSAGE and VulBERTa-GGNN this improvement is not achieved by affecting the precision, and thus, without notably increasing the number of false positives. Also in the case of graph-based representation, the model achieving the best performance is the one using a simpler world embedding layer, namely w2v + GraphSAGE, thus providing an additional evidence that the availability of samples can be the problem.

In conclusion, processing the source code as a text allows to specialize most of the recent NLP approaches, but differently from other domains there is a relevant scarcity of data as well as a greater complexity to produce good quality datasets. This problem impacts on the possibility of properly training large models. In our analysis, even using a dataset that is considerably larger than those used in some recent analysis, we have been not able to obtain a noteworthy performance improvement of state-of-the-art Transformers when working on real data. Nevertheless, structural information provided by graph-based representation can help to mitigate this problem by allowing to obtain more effective deep learning models, but further analysis are required.

5. Conclusions

In this paper, we compare different deep-learning methods for predicting if a C/C++ function is vulnerable. We have meticulously prepared a dataset of source code snippets by merging four publicly available datasets of functions from well-known open-source projects. Due to the different labeling and collection methods of each dataset, we cleaned and post-processed the samples to eliminate biases and achieve a uniform labeling. We used the samples of three dataset to train the models, reserving those belonging to the ReVeal dataset for testing, to ensure comparable and significant results in a realistic system setup of vulnerability detection.

This is the first evaluation of deep-learning techniques on such a large dataset of source code derived from real software projects. The analysis confirmed the complexity of the task by highlighting the ineffectiveness of baseline methods and showed that using structural representations of source code can lead to more accurate models.

Further in-depth analysis are necessary to assess deep-learning methods' robustness, reliability and accuracy. This will involve exploring different approaches and expanding the current dataset to make it more representative for the problem. In addition, future analysis have to evaluate the capability of deep learning methods to classify the CWEs.

References

[1] National Institute of Standards and Technology (NIST), Common Vulnerabilities and Exposures (CVE), https://nvd.nist.gov/, accessed 2023.

[2] D. A. Wheeler, Flawfinder, DWheeler, 2009. URL: https://www.dwheeler.com/flawfinder/, [Online; accessed 19-October-2023].

[3] P. Arteau, Spotbugs, SpotBugs, 2023. URL: https://spotbugs.github.io, [Online; accessed 19 october 2023].

[4] T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (2002) 654–670.

[5] S. Woo, S. Kim, H. Lee, H. Oh, Vuddy: A scalable approach for vulnerable code clone discovery, in: IEEE Symposium on Security and Privacy (SP), IEEE, San Jose, CA, USA, 2017, pp. 595–614.

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

[7] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and discovering vulnerabilities with code property graphs, in: 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 590–604.

[8] K. A. Farris, A. Shah, G. Cybenko, R. Ganesan, S. Jajodia, Vulcon: A system for vulnerability prioritization, mitigation, and management, ACM Transactions on Privacy and Security 21 (2018) 1–28. doi:10.1145/3196884.

[9] N. Alexopoulos, S. M. Habib, S. Schulz, M. Mühlhäuser, The tip of the iceberg: On the merits of finding security bugs, ACM Transactions on Privacy and Security 24 (2020) 1–33. doi:10.1145/3406112.

[10] R. Croft, D. Newlands, Z. Chen, An empirical study of rule-based and learning-based approaches for static application security testing, in: Association for Computing Machinery, New York, NY, USA, 2021, pp. 1–12.

[11] G. Lin, S. Wen, Q.-L. Han, J. Zhang, Y. Xiang, Software vulnerability detection using deep neural networks: A survey, Proceedings of the IEEE 108 (2020) 1825–1848.

[12] J. Wang, M. Huang, Y. Nie, J. Li, Static analysis of source code vulnerability using machine learning techniques: A survey, in: 2021 4th International Conference on Artificial Intelligence and Big Data (ICAIBD), IEEE, 2021, pp. 76–86.

[13] Z. Li, et al., Vuldeepecker: A deep learning-based system for vulnerability detection, in: Proc. NDSS, 2018, pp. 1–15.

[14] H. Wang, G. Ye, Z. Tang, S. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, Z. Wang, Combining graph-based learning with automated data collection for code vulnerability detection, IEEE Transactions on Information Forensics and Security 16 (2021) 1943–1958.

[15] C. D. Sestili, W. S. Snavely, N. M. VanHoudnos, Towards security defect prediction with ai, https://apps.dtic.mil/sti/pdfs/AD1090852.pdf, 2018. [Online; accessed 20 november 2023].

[16] D. Votipka, R. Stevens, E. Redmiles, J. Hu, M. Mazurek, Hackers vs. testers: A comparison of software vulnerability discovery processes, in: Proc. IEEE Symp. Secur. Privacy (SP), IEEE, 2018, pp. 374–391.

[17] H. Hanif, S. Maffeis, VulBERTa: Simplified source code pre-training for vulnerability detection, in: Proceedings of the International Joint Conference on Neural Networks, 2022.

[18] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, in: Advances in Neural Information Processing Systems, 2019, pp. 10197–10207.

[19] M. Allamanis, Graph Neural Networks on Program Analysis, Graph Neural Networks: Foundations, Frontiers, and Applications, 2021.

[20] S. Chakraborty, R. Krishna, Y. Ding, B. Ray, Deep learning based vulnerability detection: Are we there yet?, IEEE Transactions on Software Engineering 48 (2022) 3280–3296.

[21] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser, B. Ray, VELVET: a noVel Ensemble learning approach to automatically locate VulnErable sTatements, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, pp. 959–970.

[22] D. Hin, A. Kan, H. Chen, M. A. Babar, Linevd: Statement-level vulnerability detection using graph neural networks, in: Proceedings - 2022 Mining Software Repositories Conference, MSR 2022, 2022, pp. 596–607.

[23] L. Li, S. H. H. Ding, Y. Tian, B. C. M. Fung, P. Charland, W. Ou, L. Song, C. Chen, VulANalyzeR: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution, ACM Trans. Priv. Secur. 26 (2023).

[24] M. Allamanis, M. Brockschmidt, M. Khademi, Learning to represent programs with graphs, in: 6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings, 2018.

[25] J. Devlin, M. Chang, K. Lee, K. Toutanova, BERT: Pre-training of deep bidirectional Transformers for language understanding, 2018. URL: https://arxiv.org/abs/1810.04805. arXiv:1810.04805.

[26] J. Fan, Y. Li, S. Wang, T. N. Nguyen, A c/c++ code vulnerability dataset with code changes and cve summaries, in: 2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR), 2020, pp. 508–512. doi:10.1145/3379597.3387501.

[27] Y. Chen, Z. Ding, L. Alowain, X. Chen, D. Wagner, DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection, in: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, RAID '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 654–668.

[28] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, 2013. arXiv:1301.3781.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, K. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, in: I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems, volume 30, Curran Associates, Inc., 2017.

[30] D. Zou, S. Wang, S. Xu, Z. Li, H. Jin, μVulDeePecker: A deep learning-based system for multiclass vulnerability detection, IEEE Transactions on Dependable and Secure Computing 18 (2021) 2224–2236.

[31] Y. Liu, M. Ott, N. Goyal, J. D., M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, V. Stoyanov, RoBERTa: A robustly optimized BERT pretraining approach, 2019. arXiv:1907.11692.

[32] B. Fluri, M. Wursch, M. PInzger, H. Gall, Change distilling:tree differencing for fine-grained source code change extraction, IEEE Transactions on Software Engineering 33 (2007) 725–743. doi:10.1109/tse.2007.70731.

[33] F. Allen, Control flow analysis, in: Proceedings of a Symposium on Compiler Optimization, Association for Computing Machinery, New York, NY, USA, 1970, p. 1–19.

[34] J. Ferrante, K. J. Ottenstein, J. D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9 (1987) 319–349.

[35] D. Conte, P. Foggia, C. Sansone, M. Vento, Thirty years of graph matching in pattern recognition, International Journal of Pattern Recognition and Artificial Intelligence 18 (2004) 265–298. doi:10.1142/s0218001404003228.

[36] P. Foggia, G. Percannella, M. Vento, Graph matching and learning in pattern recognition in the last 10 years, International Journal of Pattern Recognition and Artificial Intelligence 28 (2014) 1450001. doi:10.1142/s0218001414500013.

[37] M. Vento, A long trip in the charming world of graphs for pattern recognition, Pattern Recognition 48 (2014) 291–301. doi:10.1016/j.patcog.2014.01.002.

[38] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, P. S. Yu, A comprehensive survey on graph neural networks, IEEE Transactions on Neural Networks and Learning Systems 32 (2021) 4–24. doi:10.1109/tnnls.2020.2978386.

[39] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, Codebert: A pre-trained model for programming and natural languages, in: Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547.

[40] Joern Team, Joern: The bug hunter's workbench, 2023. URL: https://joern.io/, [Online; accessed 30 december 2023].

[41] Y. Kim, Convolutional neural networks for sentence classification, arXiv preprint arXiv:1408.5882 (2014).

[42] W. Hamilton, Z. Ying, J. Leskovec, Inductive representation learning on large graphs, in: I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems, volume 30, Curran Associates, Inc., 2017.

[43] Q. Wang, Q. Qian, Malicious code classification based on opcode sequences and textcnn network, Journal of Information Security and Applications 67 (2022) 103151. doi:10.1016/j.jisa.2022.103151.