Evaluating Deep Learning-based Vulnerability Detection Models on Realistic Datasets

by

Krishna Kanth Arumugam

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

Abstract

The impact of software vulnerabilities on daily-used software systems is alarming. Despite numerous proposed deep learning-based models to automate vulnerability detection, the detection of software vulnerabilities remains a significant issue. While some techniques report high precision/recall scores of up to 99%, our experience leads us to believe that these models may underperform in realistic settings, specifically when evaluating vulnerability detection models on the entire source code repository of a project.

Therefore, in this thesis, we create a more comprehensive vulnerability detection dataset (i.e., *Comp-Vul*), which aims to accurately represent the realistic settings where vulnerability detection models are deployed. Then, we evaluate the performance of two *state-of-the-art* deep learning-based models, LineVul and DeepWukong, on the *Comp-Vul* dataset. Our results show that the performance of both models drops drastically, with precision dropping by 86% - 95% and F1 score dropping by 88% - 91%. Our further investigation shows that the ratio of vulnerable to non-vulnerable samples in the evaluation dataset significantly impacts the performance metrics of these models. When we visualize the embeddings produced by the models, we find that there is a substantial overlap between vulnerable and non-vulnerable samples. This shows that these models have difficulty distinguishing between vulnerable and non-vulnerable samples in the *Comp-Vul* dataset, resulting in a high number of false positives. We introduce a new program slice-level vulnerability detection technique named *SliceVul*, which leverages the powerful capabilities of Transformers and incorporates the semantic properties of source code programs such as data and control flow information. Our approach outperforms the existing *state-of-the-art* program slice-level vulnerability detection model, DeepWukong, when evaluated on the *Comp-Vul* dataset. Our study argues that accurately identifying vulnerabilities using deep learning remains a challenging task that requires improved approaches to model evaluation and design. Further research and development, complemented by realistic evaluation datasets, is required to enhance the performance of these methods.

Acknowledgements

I would like to thank my supervisor, Professor Meiyappan Nagappan, for his support and guidance throughout my Master's program. I express my gratitude to Mahmoud Alfadel for his time, support, and ideas. I would also like to thank the readers of my thesis, Professor Chengnian Sun and Professor Meng Xu, for taking the time to review my work. I must also acknowledge other researchers from various publications; this work would not have been possible without their contributions.

Table of Contents

List of Figures

Chapter 1

Introduction

Software vulnerabilities have become a significant security concern due to the increasing complexity of software and the widespread use of open-source software and third-party components. Identifying and addressing vulnerabilities in complex systems with multiple interconnected components can be a challenging task, making a comprehensive and systematic approach necessary. In recent years, machine learning models based on deep neural networks (DNNs) have shown significant promise in identifying software vulnerabilities automatically. However, the reliability of these models in detecting vulnerabilities in the real world depends on the evaluation methodology and dataset. Biases that can affect model performance can arise from various sources, such as the way data is generated and labeled. The generalizability of a model may be limited if the dataset contains implicit biases. For instance, synthetic datasets such as SARD[1] are created through automated techniques like fuzzing or genetic algorithms and are extensively used to evaluate the effectiveness of deep learning models in detecting software vulnerabilities [7, 64, 38]. Despite the existence of real-world datasets (e.g., Devign [67], Big-Vul [13], and *REVEAL* [3]), these datasets only include vulnerable and non-vulnerable samples extracted from vulnerability-related commits and represent a limited subset of the project code. The use of such datasets can limit the generalizability of the models when deployed in real-world systems.

Therefore, it is crucial to evaluate deep learning models using realistic and representative datasets to provide a more accurate evaluation of their effectiveness in detecting vulnerabilities. In this thesis, we create *Comp-Vul*, a new vulnerability detection dataset that aims to accurately depict the realistic settings in which vulnerability models would

[1]https://samate.nist.gov/SARD

be used. Our *Comp-Vul* dataset differs greatly from other existing datasets in that it includes the entire project source code representing a realistic setting. In a realistic setting, vulnerability detection models are used to scan the whole project repository to find vulnerabilities.

To demonstrate the performance of deep learning models on the realistic dataset (i.e., the *Comp-Vul* dataset), we evaluate two *state-of-the-art* models (i.e., DeepWukong [7] and LineVul [18]). Initially, we train and evaluate these models using their original datasets, i.e., the SARD[1] and Big-Vul datasets [13], respectively. The models achieve notably high precision (87% - 96%) and F1 (90% - 93%) scores. However, when we evaluate these models using the *Comp-Vul* dataset, we surprisingly observe a substantial drop in performance, with precision dropping by 86% - 95% and F1 scores dropping by 88% - 91%.

We perform further analysis to understand why these models produce a large number of false positives when evaluated on the *Comp-Vul* dataset. Our analysis shows that the ratio of vulnerable to non-vulnerable samples in the evaluation dataset greatly impacts the performance metrics of these models, especially the precision and F1 scores. As the ratio increased, the performance of these models decreased accordingly. When we visualize the embeddings generated by these models for the vulnerable and non-vulnerable samples in a two-dimensional space, we find that these models have difficulty in creating a clear distinction between the two classes, leading to incorrect identification of vulnerabilities. For example, models like LineVul, which primarily rely on lexical relationships between tokens, face challenges in accurately modeling the intricate nature of vulnerabilities in real-world software, leading to a high number of false positives.

We also propose a new Deep Learning-based program slice-level vulnerability technique called *SliceVul* that integrates the strengths of Transformers and the semantic properties(data and control flow information) of source code programs to classify whether a program slice is vulnerable or not. The accuracy, precision, recall, and F1 scores of *SliceVul* are up by 10%, 9%, 18%, and 15%, respectively, compared to DeepWukong when evaluated using the *Comp-Vul* dataset.

In summary, the main contributions of this thesis are the following:

- We introduce a new realistic vulnerability detection dataset called *Comp-Vul*, which can be used by the research community to evaluate and build vulnerability detection models more effectively.
- We evaluate the performance of two latest state-of-the-art vulnerability detection models on the *Comp-Vul* dataset and find a substantial decrease in the performance of these models compared to their original performance.

- We propose a new Deep Learning-based technique called *SliceVul*, that outperforms DeepWukong, a *state-of-the-art* slice-level vulnerability detection model.

- To advance experimental and modeling approaches for vulnerability detection, we release our *Comp-Vul* dataset along with the scripts used for the experiments to the public.[2]

Thesis organization. The rest of the thesis is organized as follows: Chapter 2 describes the limitation of existing datasets. Chapter 3 discusses previous studies related to our work. We describe the datasets, the models used in the study, and the Research Questions (RQs) in Chapter 4. In Chapter 5, we present the results of each RQ. Furthermore, we discuss the key reasons for the results in Chapter 6. In Chapter 7, we identify potential threats to the validity of our findings. Finally, in Chapter 8, we conclude the thesis and describe potential future works.

[2]https://git.uwaterloo.ca/kkarumug/evaluating-deep-learning-based-vulnerability-detection-models-on-realistic-datasets

Chapter 2

Limitations of Existing Datasets

As software systems become more complex and larger, the potential for vulnerabilities also increases. Therefore, it is crucial to have tools to discover and address these vulnerabilities. Generally, machine learning models have proven to be effective in understanding code and detecting vulnerabilities. However, obtaining and evaluating large datasets is challenging. Despite extensive research on vulnerability detection, there is a shortage of high-quality, real-world datasets in the field. Existing studies have relied on self-created datasets, which are often based on various criteria and may not reflect reality accurately. These datasets can be classified into three categories: Synthetic, Oracle-based, and Real-world Datasets. Next, we discuss the limitations of these datasets and the need for a realistic dataset to evaluate the effectiveness of deep learning models in vulnerability detection.

Synthetic datasets, such as SARD[1], created through automated methods like fuzzing or genetic algorithms, are widely used to assess and measure the efficacy of deep learning models in detecting software vulnerabilities. Although synthetic datasets can facilitate the swift production of numerous test cases, they are not without shortcomings in comparison to real-world datasets. Synthetic datasets may lack the precision and diversity of real-world datasets, rendering them less relevant. Additionally, synthetic vulnerabilities often possess lower levels of intricacy than their real-world counterparts, which can lead to deep learning models trained on synthetic datasets underperforming on real-world data.

Oracle-based datasets, unlike synthetic datasets, rely on third-party sources such as static analysis tools to provide labels for collected data samples. Although they offer more complexity than synthetic datasets, they may not fully represent real-world vulnerabilities

[1]https://samate.nist.gov/SARD

due to oversimplifications and isolations. The accuracy of the labeling source heavily influences the dataset's reliability, and incorrect labeling can lead to inaccurate results when training a deep learning model. Prior works (e.g., Scandariato et al. [52] and Zheng et al. [65]) used datasets generated using tools like the Fortify Static Code Analyzer and Infer. However, using these datasets to build a real-world vulnerability detection model may be problematic due to the potential for inaccurate labeling.

Real-world datasets for vulnerability detection, such as those used in Devign [67], Big-Vul [13], and *REVEAL* [3], are more diverse than synthetic or Oracle-based datasets, which are generated using the data available in issue-tracking systems and source code repositories. These datasets contain only a small set of vulnerable and non-vulnerable functions extracted from vulnerability-related commits. However, they have limitations because they do not fully represent the realistic setting where the entire source code of a project would typically be scanned, which is the scenario in which the vulnerability detection models would be utilized in practical applications. Furthermore, the distribution of vulnerable and non-vulnerable samples in the Devign dataset [67] is not reflective of the actual prevalence of vulnerable code in the real world. As the dataset contains an almost equal number of vulnerable and non-vulnerable samples, it does not accurately represent the real-world scenario where vulnerable code is relatively rare compared to non-vulnerable code.

Deep learning models (e.g., [3, 18, 7, 37, 67]) evaluated using these datasets may not reflect the real-world setting on how the model would be used. In a realistic setting, a developer would use the vulnerability detection model to scan the complete real-world source code repository to identify vulnerabilities. Therefore, it is necessary to evaluate models on whole source code repositories of several real-world projects to accurately assess their performance. Evaluating models in a controlled or simulated environment may provide overly optimistic misleading results, as real-world data is often more complex and varied than simulated or controlled data, and models may not generalize well to new or unexpected situations. Hence, realistic evaluation settings can provide a more accurate assessment of a model's performance by taking into account the complexity and variability of the data that the model will encounter in practice.

Chapter 3

Related Works

In this chapter, we discuss the related works and reflect on how they compare with ours. The works most related to our study fall into two main categories: 1) studies on vulnerability detection datasets, and 2) studies on vulnerability detection techniques.

3.1 Vulnerability detection datasets

Grahn and Zhang [19] collected different existing C/C++ vulnerabiltiy datasets (e.g., Big-Vul [13], Draper VDISC [51]) and analyzed the representativity and duplicativeness of these datasets. They showed that some of these datasets have limited usefulness for machine learning-assisted vulnerability detection. They proposed a new dataset called *Wild C* that contains 10.3 million C/C++ files from numerous open-source projects. Grahn and Zhang [19] listed down comment prediction, function name recommendation, code completion, and variable name recommendation as the potential use cases for this dataset. The drawback of this dataset is that they provide file-level data samples without any labels associated with each file. Since it does not contain any labels for the collected files, it may not be useful for building vulnerability detection classification models. This is because these models require labels for each sample to learn the vulnerability patterns accurately. Our *Comp-Vul* dataset contains the complete source code of the project along with vulnerable/non-vulnerable labels for each sample.

The Big-Vul dataset [13] is a collection of C/C++ functions from 348 open-source GitHub projects spanning from 2002 to 2019, containing 91 different Common Weakness

Enumerations (CWEs). The dataset comprises a total of 188,636 C/C++ functions, of which 10,900 have been manually verified as vulnerable, and 177,736 are verified as non-vulnerable. It has been widely used in several security vulnerability detection studies [35, 23, 18, 6]. It is worth noting that the Big-Vul dataset only includes vulnerable and non-vulnerable functions that have been extracted from vulnerability fixing commits [13]. As a result, this dataset does not capture all functions within a project, and the dataset may not represent the entire codebase. We build a more comprehensive realistic dataset called *Comp-Vul* dataset on top of the Big-Vul dataset. Our *Comp-Vul* dataset differs from Big-Vul dataset in that it includes all the source code for each of the top ten real-world projects by vulnerability counts in the Big-Vul dataset. This comprehensive approach provides a much more accurate representation of a realistic vulnerability detection setting, which is essential for training and evaluating vulnerability detection models.

3.2 Studies on vulnerability detection techniques.

Coverity [1], Fortify [2], Flawfinder [3], RATS [4], and Checkmarx [5] are some of the conventional static program analyzers for examining big software systems. These techniques rely largely on traditional static analysis theories such as data flow, abstract interpretation, and taint analysis and require human experts to create effective detection criteria. They have efficiently identified well-defined low-level problems, such as memory issues. But when it comes to detecting high-level vulnerabilities, they frequently suffer from a large number of false positives and false negatives.

There are several works on discovering vulnerabilities statically using code similarity analysis [46], [25], [27]. In most cases, code similarity analysis converts each code fragment into an abstract representation and then computes the similarity between two abstractions. They establish a similarity threshold and deem the target code susceptible if the similarity between the target code fragment and the vulnerable ones exceeds the threshold. However, these systems need the use of human specialists to identify characteristics in order to apply appropriate code similarity algorithms for various sorts of vulnerabilities [49].

Several studies explored the effectiveness of traditional machine learning techniques [41, 61, 64]. Neuhaus and Zimmermann [41] investigated the prevalence of software vulnera-

[1]https://scan.coverity.com/
[2]https://www.hpfod.com
[3]https://dwheeler.com/flawfinder/
[4]https://code.google.com/archive/p/ rough-auditing-tool-for-security/
[5]https://www.checkmarx.com

bilities in Red Hat packages using Support Vector Machines (SVM) [22]. The study analyzed the defect data from over 3,241 Red Hat packages and evaluates the effectiveness of SVM in identifying vulnerabilities in software packages. Zheng et al. [64] examined the effectiveness of different machine learning techniques, like Decision Tree [47], Random Forests [1], k-nearest neighbors (KNN) [45], and SVM, in detecting software vulnerabilities. The results of the study provided insights into the strengths and weaknesses of different machine learning techniques for vulnerability detection. Yan et al. [61] proposed a machine-learning-guided typestate analysis technique for static detection of use-after-free vulnerabilities in software. The proposed method employs a combination of static analysis, machine learning, and typestate modeling to identify use-after-free bugs in software programs accurately. Lomio et al. [39] investigated whether machine learning algorithms like SVM, KNN, Decision Tree, and Boosting algorithms [16, 17, 5] could improve the performance of Just-in-Time software vulnerability detection utilizing various software metrics (process metrics, product metrics, and text metrics) from Java project files.

Other studies focused on exploring the potential of various deep learning methods to detect vulnerabilities [32, 37, 8, 18, 67, 7, 36]. For example, Convolutional Neural Networks (CNN) [31] have been used to forecast software defects and locate defective source code [32]. Li et al. [36] made multiple kinds of deep neural networks such as CNN, LSTM [24], and GRU [9] to detect vulnerabilities using syntax and semantic information of programs. Vuldeepecker [37] detects resource management issues and buffer overflows by training an LSTM model with code embedding and data-flow information of a program. VGDetector [8] uses a control flow graph and a graph convolutional network [29] to detect control-flow vulnerabilities. Zhou et al. [67] pinpointed bugs at the method level using Graph Neural Networks and program dependence graph information of a source code program. Cheng et al. [7] used program slices along with Graph Neural Networks to develop program slice-level vulnerability detection models.

Much of the aforementioned work used either synthetic datasets (e.g., SARD), datasets created using oracles like static analysis tools (e.g., [52, 65]), or real-world datasets that do not accurately reflect the realistic settings where these models would be used for detecting vulnerabilities (e.g., [67, 13, 3]). This motivated us to create the *Comp-Vul* dataset that tackles the lack of realistic evaluation datasets in the existing techniques. Also, to the best of our knowledge, we are the first to propose a slice-level vulnerability detection technique(*SliceVul*) that uses both transformers and semantic code information(control and data flow information) of source code programs to detect vulnerabilities.

The work most closely related to ours is Chakraborty et al. [3]. Similar to our study, they proposed *REVEAL* dataset to highlight the limitations of existing deep learning-based vulnerability detection models (i.e., [37, 67, 36]). Based on their evaluation, they reported that the studied models suffer from several issues, such as the model's inability to learn relevant features, data duplication, data imbalance, and unrealistic evaluation settings. However, their evaluation still suffers from the same limitations as DeepWukong [7] and LineVul [18] models, as they only consider functions present in selected vulnerability-related commits, rather than considering all the functions within a project. Fu and Tantithamthavorn [18] proposed LineVul, a CodeBERT-based model [14] to detect vulnerabilities. Their model outperforms several techniques (e.g., [35, 67, 36, 37, 51]) including *REVEAL* [3]. This motivated us to include LineVul as one of the evaluated models in our study. Our study differs from the prior work since we evaluate two *state-of-the-art* techniques (i.e., [7], [18]) on *Comp-Vul* dataset, which reflects realistic vulnerability detection settings.

Chapter 4

Study Design

In this chapter, we describe the datasets used in our study including our *Comp-Vul* dataset (Section 4.1). Then, we present the models we evaluate using the datasets (Section 4.2). Finally, we describe the research questions driving our investigation (Section 4.3), and the evaluation metrics (Section 4.4).

4.1 Datasets

In this section, we provide an overview of the existing datasets, namely SARD [1] and Big-Vul [13], utilized in the two deep learning-based vulnerability detection techniques, DeepWukong [7] and LineVul [18] along with their inherent limitations. Also, we outline the process we undertook to generate our dataset: Comprehensive Real-World Vulnerability Detection Datasets (*Comp-Vul*).

4.1.1 SARD

The Software Assurance Reference Dataset (SARD) SARD[1]contains a vast number of artificially produced C/C++ programs containing numerous security vulnerabilities. It has been frequently used by several research works to assess the effectiveness of vulnerability detection methods [7, 37, 42, 33]. The SARD dataset labels code statements as good

[1]https://samate.nist.gov/SARD

(code statements without vulnerabilities) or bad (code statements with vulnerabilities) [7]. The limitation of the SARD dataset is that it comprises solely artificially generated vulnerabilities, which may not provide an accurate representation of real-world software vulnerabilities. Additionally, it fails to represent the real-world code written by developers. As a result, the dataset may not be able to capture the full range of complexities and variations that are found in real-world software. This may limit the ability of researchers to evaluate the effectiveness of vulnerability detection methods in a realistic setting.

4.1.2 Big-Vul

The Big-Vul dataset [13] is composed of C/C++ functions collected from 348 open-source GitHub projects spanning from 2002 to 2019, containing 91 different Common Weakness Enumerations (CWEs). It has been frequently used by several security vulnerability detection studies [35, 23, 18, 6]. The dataset comprises a total of 188,636 C/C++ functions, of which 10,900 have been manually verified as vulnerable, and 177,736 are verified as non-vulnerable. It also contains metadata of vulnerabilities such as line numbers of vulnerabilities present in a function, vulnerability fixing commit hash, CVE IDs, severity rankings, and summaries from the public Common Vulnerabilities and Exposures (CVE) database and related source code repositories. Both the vulnerable and non-vulnerable functions present in the Big-Vul dataset are extracted from vulnerability fixing commits [13]. The Big-Vul dataset has a limitation in that the vulnerability fixing commits used to identify non-vulnerable functions only cover a limited number of functions within a project rather than all the functions present in the project. In a realistic setting, a vulnerability detection model is employed to scan the entire project repository that contains all the functions of a project. The results might be misleading and elevated if the vulnerability detection models are evaluated on a dataset like Big-Vul that contains only a subset of the project functions.

4.1.3 Comp-Vul

In this study, we want to assess the performance of two recent vulnerability detection models (i.e., DeepWukong and LineVul) using a comprehensive vulnerability dataset that reflects a more realistic setting. To that end, we introduce a new dataset (i.e., *Comp-Vul*) to overcome the limitations of the existing vulnerability detection datasets by constructing a more realistic notion of vulnerability data containing real-world complex vulnerable and non-vulnerable source code programs. Our *Comp-Vul* dataset differs from other datasets like SARD or Big-Vul in that it includes all the source code for each real-world project. This comprehensive approach provides a much more accurate representation of a realistic production setting, which is essential for training and evaluating vulnerability detection models. In contrast, SARD only contains artificially synthesized samples, and Big-Vul includes only a subset of the project code. By providing a complete view of the project code, the *Comp-Vul* dataset serves a more realistic dataset for vulnerability detection model development.

Our *Comp-Vul* dataset includes a separate training and test dataset, which can be used for training and evaluating the vulnerability detection models, respectively. We employ a time-based strategy for creating the samples for the training and test datasets. This approach reflects the fact that vulnerability detection models need to be trained on historical data and then used to identify new vulnerabilities that emerge over time. By simulating this process, we aim to ensure that the models are evaluated on a dataset that reflects how they would be used in a realistic production setting. This can help to ensure that the models are detecting vulnerabilities in realistic settings before they can be deployed in a production system. We explain how we obtain the vulnerable and non-vulnerable samples present in the training and test datasets next.

To build a quality set of vulnerable samples for training and testing, we select the top ten projects in the Big-Vul dataset [13] based on the number of vulnerabilities in the projects. We extract the date on which the vulnerable functions were fixed using the vulnerable fixing commit hashes present in the Big-Vul dataset. For each project, we order the vulnerable functions based on the date on which they were fixed. We then take the first 80% of the ordered vulnerability functions for the training dataset, with the remaining 20% used for the test dataset. For instance, in the case of the FFmpeg project, the first 80% of the vulnerability functions have vulnerability fixing dates between August 3, 2013, and May 30, 2018. These functions belong to the training dataset. The remaining 20% of the data have vulnerability fixing dates between June 28, 2018, and August 5, 2019. These functions belong to the test dataset. In total, the ten projects in our dataset comprise 5,528 vulnerable functions, 4,418 functions of which we use for the training dataset and 1,110 functions for the test dataset.

To begin the process of creating non-vulnerable functions for the datasets, we first clone the remote repository of each of the ten selected projects. Then, we check out the project on two different snapshots, one for the training dataset and one for the test dataset. We take the most recent snapshots for each dataset. For instance, in our example of the FFmpeg project, we clone the FFmpeg repository on May 31, 2018, to create the non-vulnerable samples of the training dataset. Recall that the last vulnerability fix date for a vulnerable sample of the FFmpeg project in the training dataset is from May 30, 2018.

For the FFmpeg project in the test dataset, we take the second snapshot on August 6, 2019. In order to obtain a sample of non-vulnerable functions for each project, we first extract all functions from the source code files that we have collected. We calculate the MD5 hash of these collected functions and the vulnerable functions we have previously collected. We then compare the MD5 hash of the collected functions to the MD5 hash of vulnerable functions. If the MD5 hash of a collected function does not match with any of the MD5 hashes of the vulnerable functions, it is labeled as non-vulnerable. After completing this process, we end up with a total of 1,682,713 non-vulnerable functions. Out of these functions, 769,464 belong to the training dataset, and 913,249 belong to the test dataset. Note that the number of non-vulnerable functions is more in the test dataset, because they are extracted from a later version of the project compared to the version used for the training dataset.

4.2 Models Employed in the Study

In this study, we choose LineVul [18] to conduct our experiments as Fu and Tantithamthavorn [18] compare the performance of their model against other well-known vulnerability detection works [35, 3, 67, 36, 37, 51]. Furthermore, we also choose DeepWukong [7] to conduct our experiments as it reports *state-of-the-art* performance (up to 99% F1 score) in detecting vulnerabilities at the program slice level using program semantics and graph neural networks.

4.2.1 DeepWukong

DeepWukong [7] is a Deep Learning-based model built using a Graph Neural Network (GNN) architecture [2], which takes *XFGs* as inputs and classifies them as vulnerable or non-vulnerable *XFG*. *XFGs* are subgraphs extracted using the Program Dependence Graphs (PDGs) [15] of the source code programs. PDGs are directed graphs where each node in the graph represents a code statement in the source code, and each edge in the graph represents a data or control flow dependence between two code statements. *XFGs* are generated by performing forward and backward slicing [59] starting from a program point of interest, such as an API call or an Arithmetic Expression in the PDG. If an *XFG* contains at least one vulnerable code statement, it is labeled as vulnerable; otherwise, it is labeled as non-vulnerable. The vulnerable and non-vulnerable *XFGs* are fed as inputs to a Graph Neural Network (GNN) to train an effective vulnerability detection model.

4.2.2 LineVul

LineVul [18] is a Deep Learning-based model built using CodeBERT [14], which takes in a chunk as input and classifies it as a vulnerable or non-vulnerable *chunk*. A chunk is a sequence of code tokens generated from source code programs. First, the source code statements in the programs are tokenized using the Byte Pair Encoding (BPE) subword tokenization technique [53]. Then, the tokenized source code programs are split into chunks of up to 512 tokens, which is the maximum input size accepted by the CodeBERT model. If a chunk contains a vulnerable line, it is labeled as vulnerable; otherwise, it is labeled as non-vulnerable. The code tokens in the chunks are converted to embedding vectors so that they can be fed as input to the CodeBERT model. An embedding vector is a combination of a word encoding vector and a positional encoding vector. The CodeBERT model is made up of several multi-head self-attention layers and a feed-forward neural network that is fully connected. Finally, the output vector from the CodeBERT is passed into a Dense neural network layer, which classifies whether a chunk is vulnerable or non-vulnerable.

4.3 Research Questions

In this section, we introduce our research questions by explaining the motivation behind each one.

RQ$_1$: *To what extent can the findings of the DeepWukong and LineVul models be replicated to confirm the results reported in the original studies?*

Our study aims to evaluate the performance of the DeepWukong and LineVul models in a more realistic setting. As a first step towards our goal, in this RQ, we run the DeepWukong and LineVul models on the SARD and the Big-Vul datasets, respectively, and verify the findings reported in the original papers. By doing so, we aim to ensure that the results previously reported are reliable and can be replicated. Additionally, we use the results obtained from these experiments as our baseline models, which we will compare to the performance of the models in our introduced realistic settings.

RQ$_2$: *How do the DeepWukong and LineVul models perform in a realistic evaluation setting compared to the evaluation setting used in the original studies?*

DeepWukong and LineVul are two *state-of-the-art* vulnerability detection models that have shown promising results in detecting security vulnerabilities. However, we argue that the datasets with which these models have been evaluated may not reflect the realistic setting for detecting security vulnerabilities. First, the DeepWukong model extensively used the SARD dataset for evaluation. The SARD dataset contains artificially generated samples, which may not represent the complexities present in real-world vulnerabilities. Second, the dataset used for evaluation contains vulnerable and non-vulnerable samples taken from the vulnerability-related commits, i.e., the non-vulnerable sample is limited to the source code that is present in the vulnerability-related commits. A more realistic setting for evaluation should consider the remaining source code (code not touched by the vulnerability-related commits) for non-vulnerable samples. Similarly, the LineVul model uses the Big-Vul dataset [13], which suffers from the same problem where the dataset used for evaluation contains a limited number of non-vulnerable samples.

In summary, these evaluation settings do not accurately reflect how the vulnerability detection models would be used in the realistic setting where the models would scan the entire project source code files. Therefore, it is crucial to evaluate these models on a dataset that accurately reflects a more realistic evaluation setting to assess their performance in practical applications. In this RQ, we evaluate the DeepWukong and LineVul models on our *Comp-Vul* test dataset.

RQ$_3$: *How do the DeepWukong and LineVul models perform in a realistic evaluation setting when trained using a similar realistic training dataset?*

In RQ$_2$, we use *Comp-Vul* dataset to evaluate the models. However, we recognize that the training dataset used to train these models (i.e., Big-Vul) may not be representative of the same distribution as the *Comp-Vul* dataset used for evaluation. Doing so can lead to poor model performance as the training and test datasets are from different distributions. The training and test datasets are usually representative of the real-world data where the model will be used to perform its task. Hence, it is crucial for both datasets to be selected from the same distribution as the data in which the model will be used. Therefore, if the training and test datasets are from the same distribution, the models may achieve better results. In this RQ, we aim to investigate the impact of data distribution on the performance of these models by training and testing them using our *Comp-Vul* dataset. By doing so, we hope to determine whether using a training dataset that is representative of the same distribution as the evaluation dataset can lead to better model performance.

4.4 Evaluation metrics.

The evaluation metric should be able to accurately measure how well the model performs on the task at hand. Vulnerability prediction is a binary classification task. One of the most popular metrics used for binary classification is accuracy. However, it does not take into account false positives and false negatives. Hence, we evaluate the models using additional metrics (i.e., precision, recall, and F1-score) [43].

Precision is the fraction of the vulnerabilities detected by the model that are actually vulnerabilities, i.e., it measures often samples classified as vulnerable are truly vulnerable. For example, in the case of the DeepWukong model, it will measure how many XFGs that are predicted as vulnerable are truly vulnerable XFGs. The number of false positives (falsely classified as vulnerable) affects the precision score greatly. If the precision score is very low, it means the model is incorrectly predicting many non-vulnerable samples as vulnerable resulting in a huge number of high false positives. A high precision means that when the model detects a vulnerability, there is a high probability that it is a real vulnerability.

The recall is the fraction of the actual vulnerabilities in a system that are detected by the model, i.e., it measures how effective our model is at detecting vulnerabilities. For example, in the case of the DeepWukong model, it will measure how many true vulnerable XFGs are correctly identified as vulnerable XFGs by the model. The number of false negatives(falsely classified as non-vulnerable) affects the recall score greatly. A high recall means that the model is able to detect most of the vulnerabilities correctly, and the number of false negatives is low. F1-score combines precision and recall to provide an overall picture of how well these two measurements are balanced.

Chapter 5

Experiments & Results

In this chapter, we present our experimental results with respect to each (RQ).

5.1 Research Question 1

Question: *To what extent can the findings of the DeepWukong and LineVul models be replicated to confirm the results reported in the original studies?*

Approach: We first download the SARD dataset [1] and the Big-Vul dataset [2] used in DeepWukong [7] and LineVul [18] studies, respectively. We create the DeepWukong model inputs (XFGs) and the LineVul model inputs (chunks) using the SARD dataset and the Big-Vul dataset samples, respectively. We obtain a total of 14,261 vulnerable XFGs, 74,963 non-vulnerable XFGs from the Big-Vul dataset, and 9,733 vulnerable chunks, 216,568 non-vulnerable chunks from the LineVul dataset. The generated XFG and the chunk datasets are split into training and test datasets where 80% of the samples belong to the training dataset and the remaining 20% samples belong to the test dataset. Following the author's original experiments, we train the DeepWukong model for 50 epochs and the LineVul model for 10 epochs. We evaluate the models using the test datasets.

[1]https://github.com/jumormt/DeepWukong
[2]https://github.com/awsm-research/LineVul

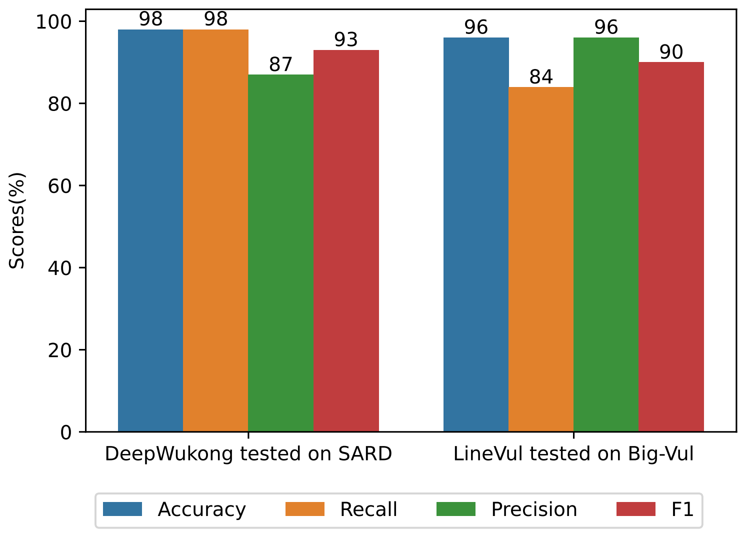


Figure 5.1: Replication results of DeepWukong and LineVul models

Results: Figure 5.1 shows the replication results of the DeepWukong and LineVul models. We can see that the DeepWukong model achieves 98%, 87%, 98%, and 93%, for accuracy, precision, recall, and F1-score, respectively. The accuracy and F1-score for the DeepWukong model differ by +1% and -2%, respectively, from the results reported by the DeepWukong authors.

Our results for the LineVul model follow a similar trend. The accuracy, precision, recall, and F1-score for the LineVul model are 96%, 96%, 84%, and 90%, respectively. Comparing with the results reported by the authors of LineVul, we find that the precision, recall, and F1-score differ by -1%, -2%, and -1%, respectively.

Overall, we can observe that the differences in the metrics are negligible. The small difference in the metrics can be because of the presence of different samples in the training/test datasets. When a dataset is split into a training set and a test set, the samples in the two sets are chosen randomly. This means that the samples in the training set and the test set will be different each time we split the dataset. As a result, the model's performance on the test set may vary from one split to the next. The model's performance on the test set may depend on the specific samples that are included in the test set. For example, if the test set contains a particularly difficult or easy sample, this can affect the model's overall performance on the test set.

ANSWER: Using the same methods as the DeepWukong and LineVul studies, we were able to replicate their results for the SARD and Big-Vul datasets.

5.2 Research Question 2

Question: *How do the DeepWukong and LineVul models perform in a realistic evaluation setting compared to the evaluation setting used in the original studies?*

Approach: In this RQ, we evaluate the DeepWukong and the LineVul models using the *Comp-Vul* test dataset described in section 4.1.3. We use the LineVul model trained in RQ_1 for the evaluation, i.e., we train using the same method as the original LineVul study while using *Comp-Vul* test dataset for evaluation. The DeepWukong model in RQ_1 is trained using SARD, which contains artificially created samples. Since our *Comp-Vul* test dataset contains complex real-world samples, we train a new DeepWukong model using the Big-Vul dataset for evaluation. First, we generate the DeepWukong model inputs (XFGs) using the Big-Vul dataset. We obtain a total of 28,294 vulnerable XFGs and 639,047 non-vulnerable XFGs. Similar to RQ_1, we train the DeepWukong model for 50 epochs using the generated XFGs. To evaluate the trained models (DeepWukong and LineVul) using the *Comp-Vul* test dataset, we first generate the model inputs (XFG and chunks) for the *Comp-Vul* test dataset. We obtain a total of 5,386 vulnerable XFGs, 4,844,728 non-vulnerable XFGs, 1,316 vulnerable chunks, and 1,020,801 non-vulnerable chunks. These samples are then evaluated using the respective trained models.

Results: Figure 7.2 shows the results of the performance of the DeepWukong and LineVul models, trained using the Big-Vul dataset and evaluated with the *Comp-Vul* test dataset. For the DeepWukong model, we achieve an accuracy, precision, recall, and F1-score of 91%, 1%, 87%, and 2%, respectively. When comparing these results to those obtained in RQ_1 (i.e., compared to the results of the DeepWukong model trained and tested using the SARD dataset), we observe a decrease of 7%, 86%, 11%, and 91% in accuracy, precision, recall, and F1-score, respectively.

As for the LineVul model, we obtain an accuracy, precision, recall, and F1-score of 89%, 1%, 90%, and 2%, respectively. In comparison to the results obtained in RQ_1 (i.e., compared to the results of the LineVul model trained and tested using the Big-Vul dataset), we observe a decrease of 7%, 95%, and 88% in accuracy, precision, and F1-score, respectively. However, we also observe an increase of 6% in recall.

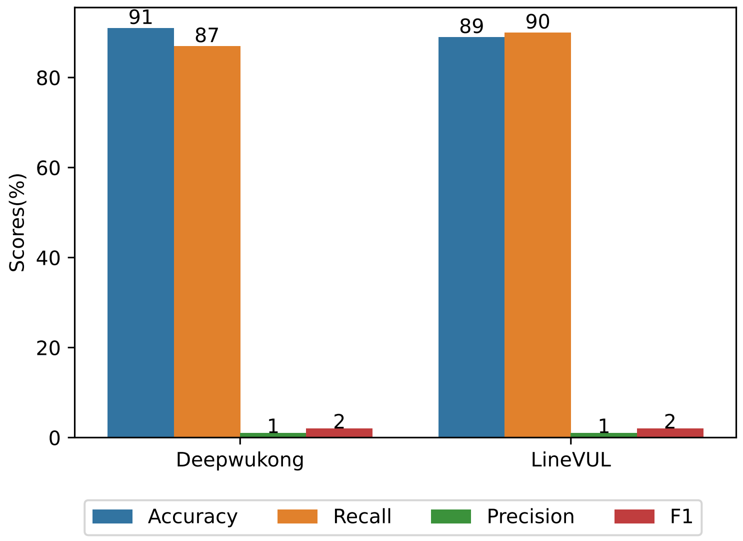


Figure 5.2: Results of the DeepWukong and LineVul models trained using the Big-Vul dataset and evaluated using the *Comp-Vul* test dataset

Overall, our findings reveal a substantial decrease in precision for both the DeepWukong and LineVul models. This suggests that the predictions made by these models contain a considerable number of false positives, which can have a huge impact on the effectiveness of vulnerability detection. Specifically, the DeepWukong model generated 439,494 false positives, while the LineVul model produced 114,629 false positives. Such a large number of false positives renders these models unusable.

Our study underscores the importance of evaluating vulnerability detection models using datasets that reflect realistic settings. In this regard, our results demonstrate the critical role played by the *Comp-Vul* dataset in understanding the true capabilities of such models. By utilizing this dataset, we gain a more accurate understanding of the performance of vulnerability detection models and can identify areas that require improvement.

ANSWER: Existing vulnerability detection models, such as DeepWukong and LineVul, can produce a high number of false positives when tested using datasets that represent more accurate real-world testing settings.

5.3 Research Question 3

Question: *How do the DeepWukong and LineVul models perform in a realistic evaluation setting when trained using a similar realistic training dataset?*

Approach: In the previous RQ, we evaluate the effectiveness of our models using the *Comp-Vul* dataset for evaluation only. However, in this question, we use the *Comp-Vul* dataset for both model training and evaluation. The training dataset consisted of 2,699,492 XFGs (12,574 vulnerable XFGs and 2,686,918 non-vulnerable XFGs) and 887,455 chunks (5,213 vulnerable chunks and 882,242 non-vulnerable chunks). This dataset is highly imbalanced, with a disproportionate number of non-vulnerable samples compared to vulnerable ones. Hence, we also investigate the impact of this imbalance on model performance by training additional models on balanced datasets. We randomly select non-vulnerable samples equal to the number of vulnerable samples present. We obtain a balanced XFG dataset of size 25,148 (12,574 vulnerable XFGs + 12,574 non-vulnerable XFGs) and a balanced chunk dataset of size 10,426 (5,213 vulnerable chunks + 5,213 non-vulnerable chunks). Overall, we train a total of four models (one imbalanced-trained-DeepWukong, one balanced-trained-DeepWukong, one imbalanced-trained-LineVul, and one balanced-trained-LineVul) using the same training parameters used in RQ_1. Finally, we evaluate the trained DeepWukong and LineVul models using the same XFG and chunks test dataset used in RQ_2 (i.e., *Comp-Vul* test dataset).

Results: Figure 5.3 shows the results of the DeepWukong and LineVul models that are trained using imbalanced datasets (XFGs and chunks). The accuracy is 99% for both the models, while the precision, recall, and F1 scores are 0%. The precision, recall, and F1-scores are extremely down for both the models compared to the results obtained in RQ_1 and RQ_2. The reason for the high accuracy score is because of the imbalanced nature of the XFG and chunk datasets. Due to the large number of non-vulnerable samples in the datasets, the model consistently predicts that a sample is non-vulnerable, leading to a high accuracy score but lower recall, precision, and F1 scores. The models struggle to learn the patterns in the minority samples(vulnerable samples) because the majority samples(non-vulnerable samples) dominate the training process. The reason for this behavior is that the models are optimized to minimize the loss function, which penalizes incorrect predictions. In our imbalanced dataset, the non-vulnerable class has more samples than the vulnerable class. As a result, the models learn to always classify a sample as non-vulnerable to minimize the loss function, even if it is not the correct prediction.

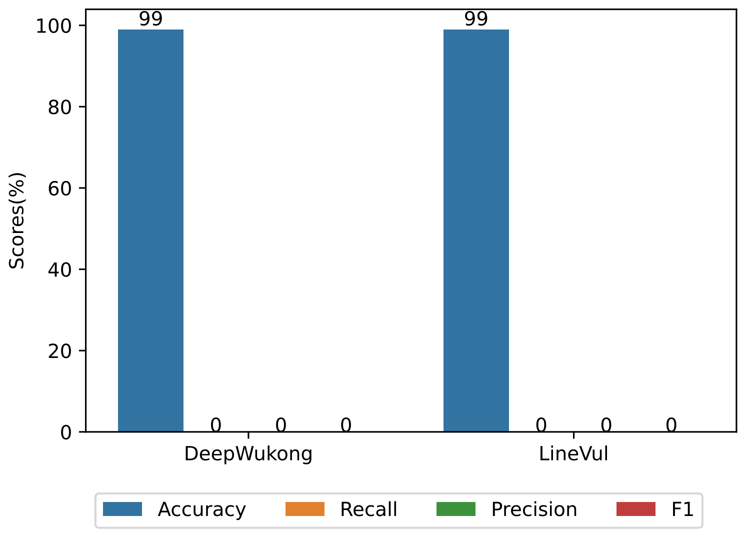


Figure 5.3: Results of the DeepWukong and LineVul models that are trained using the imbalanced dataset(*Comp-Vul* dataset) and evaluated using the *Comp-Vul* test dataset

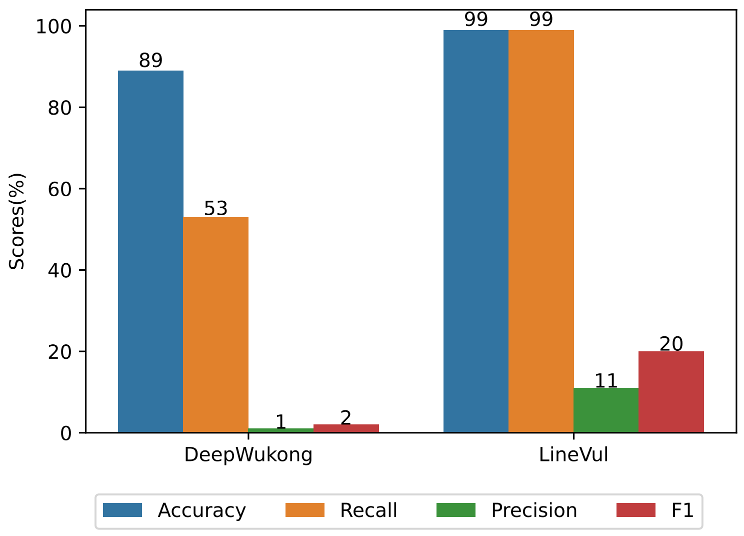


Figure 5.4: Results of the DeepWukong and LineVul models that are trained using the balanced dataset(*Comp-Vul* dataset) and evaluated using the *Comp-Vul* test dataset

Figure 5.4 shows the performance of the DeepWukong and LineVul models that are trained using balanced datasets (XFGs and chunks). The results indicate that the Deep-

Wukong model trained on the balanced dataset performs with 89% accuracy, 1% precision, 53% recall, and 2% F1 score, which is down by 9%, 86%, 45%, and 91%, respectively, compared to the DeepWukong model evaluated in RQ_1. The LineVul model trained on the balanced dataset performs with 99% accuracy, 11% precision, 99% recall, and 20% F1 score. The precision and F1 scores are down by 85% and 70%, respectively, compared to the LineVul model evaluated in RQ_1.These results suggest that these models produce a high number of false positives, even when trained using a dataset that is similar to the realistic evaluation dataset.

Furthermore, the recall score of the DeepWukong model has decreased by 34% compared to RQ_2, while for the LineVul model it has increased by 9%. The precision and F1 scores for the DeepWukong model remain the same, while for the LineVul model, they have increased by 10% and 18%. This variation in the performance metrics between the DeepWukong and LineVul models can be due to several factors, such as model input structure (XFG vs. chunks), model architecture (GNN vs. CodeBERT), and more.

Additionally, the study shows that training the models using a balanced dataset leads to a considerable improvement in their ability to identify vulnerable samples in the test dataset. The DeepWukong model's recall, precision, and F1 scores are up by 53%, 1%, and 2%, respectively, when trained using a balanced dataset compared to an imbalanced dataset. A similar pattern is observed for the LineVul model. However, the models still produce a high number of false positives, similar to the models evaluated in RQ_2.

ANSWER: Despite training the DeepWukong and LineVul models with a realistic training dataset that closely resembles the realistic test dataset, we observe that these models still generate a high number of false positives

Chapter 6

Discussion

In this chapter, we investigate a number of problems that LineVul and DeepWukong models suffer from, which can explain some reasoning behind the failure on the *Comp-Vul* dataset.

6.1 Class Imbalance of Evaluation Dataset

Real-world datasets often suffer from imbalanced class distributions where the number of non-vulnerable examples enormously outweighs the number of vulnerable ones. This class imbalance can lead to models being biased towards the majority class, resulting in poor performance on vulnerable examples. Therefore, we investigate the impact of class imbalance on the performance metrics of the LineVul and DeepWukong models in the context of vulnerability detection.

Approach: In RQ_2, we observe extremely low precision and recall scores for both studied models. To understand the influence of class balance on model performance, we conduct an experiment using the *Comp-Vul* dataset from RQ_2. We create test datasets by varying the ratio of vulnerable to non-vulnerable samples using the XFG and chunk samples of the *Comp-Vul* test dataset. The ratio between vulnerable to non-vulnerable samples is 899 for the XFG dataset and 775 for the chunk dataset. We generate 900 XFG datasets (ranging from 0 - 899) and 776 chunk datasets (ranging from 0 - 775) by randomly selecting non-vulnerable samples based on the ratio numbers. The vulnerable samples are kept constant across all datasets.

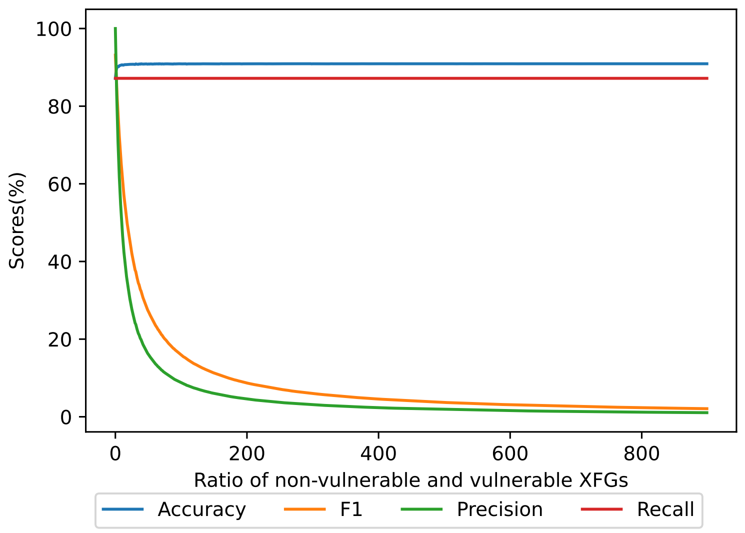


Figure 6.1: Performance of the DeepWukong model evaluated on *Comp-Vul* test dataset for different ratios of the non-vulnerable and vulnerable XFGs.

However, for non-vulnerable samples, we randomly select them based on the desired ratio and the count of vulnerable samples in each dataset. As an example, in the XFG dataset, we have 5,386 vulnerable XFGs and 4,844,728 non-vulnerable XFGs. To construct the test XFG dataset with a ratio of 2, we randomly select 10,772 non-vulnerable XFGs (2 times the number of vulnerable XFGs) from the 4,844,728 non-vulnerable XFGs, while keeping all 5,386 vulnerable XFGs. This results in a total of 16,158 XFGs (5,386 vulnerable XFGs + 10,772 non-vulnerable XFGs) in the test XFG dataset with a ratio of 2. We evaluate the DeepWukong and LineVul models on each dataset using the same experimental setup from RQ_2 repeated 100 times with different randomly sampled datasets each time to reduce sampling bias. We report the mean performance metrics obtained from running the experiment 100 times to obtain more reliable metrics and reduce the variability of the results.

Results: Figures 6.1 and 6.2 present line plots depicting the performance metrics of LineVul and DeepWukong models when tested on the *Comp-Vul* dataset. The x-axis of the line plots represents the ratios between non-vulnerable and vulnerable samples, while the y-axis represents the performance scores in percentage for each metric.

As we can see from the figures, the recall metric remains constant across all ratios since the number of vulnerable samples is constant, resulting in a steady number of false

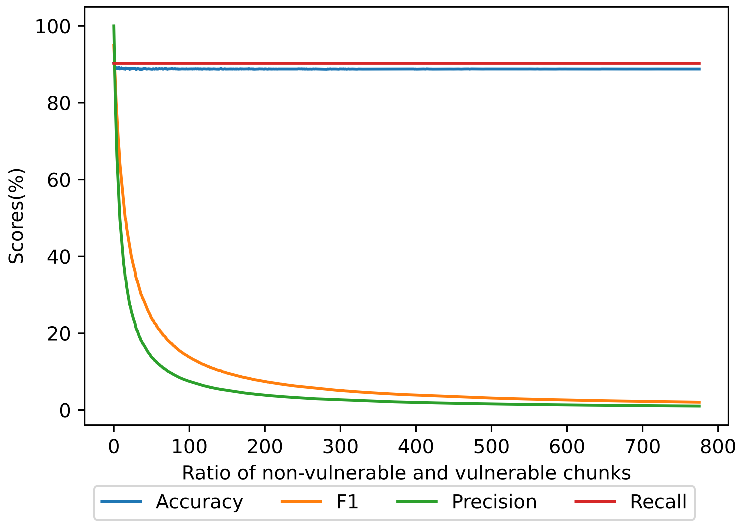


Figure 6.2: Performance of the LineVul model evaluated on *Comp-Vul* test dataset for different ratios of the vulnerable and non-vulnerable chunks.

negatives and true positives. However, the precision and F1-score show a decreasing trend with an increase in the ratio. The precision and F1 scores sharply decrease until the 600th ratio for DeepWukong model and 500th ratio for LineVul model, after which the recall and F1-score begin to flatten around the 1% and the 2% region for both models. The accuracy metric has minimal variations, which are only visible in the fourth decimal place of the percentage values, making them almost invisible in the line plots. These differences are negligible and can be considered insignificant in the context of the experiment. In general, an increase in non-vulnerable samples in the dataset results in a substantial increase in false positives.

Our findings highlight the criticality of accounting for class imbalance in vulnerability detection models to minimize the risk of erroneously labeling non-vulnerable samples as vulnerable and incurring a high number of false positives. By evaluating the performance of vulnerability models on a realistically imbalanced dataset such as *Comp-Vul* and conducting a thorough analysis, developers can confidently use these models for vulnerability detection tasks.

6.2 Insufficient Class Separation

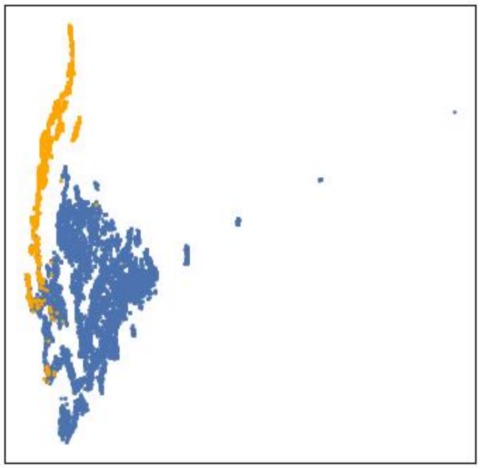
LineVul and DeepWukong are models that transform source code into fixed-size embeddings, which are utilized to train a neural network to classify vulnerable and non-vulnerable samples accurately. The effectiveness of these models in vulnerability detection is influenced by the degree to which the embeddings of the vulnerable and non-vulnerable classes are distinct and separable. The greater the distinction and separability of the embeddings, the more straightforward it is for the model to differentiate between the two classes. To investigate this, we perform an experiment to visualize the ability of the models under examination to clearly separate the vulnerable and non-vulnerable class samples.

Approach: We employ t-distributed Stochastic Neighbor Embedding (t-SNE) [56] to visualize the embeddings produced by the DeepWukong and LineVul models. t-SNE is a powerful visualization tool for high-dimensional data that aids in the identification of clusters and patterns by reducing data dimensionality while preserving local structure. First, we extract the embeddings from the DeepWukong and LineVul models. The *[CLS]* embedding vectors generated by the CodeBERT model represent the embeddings for the LineVul model, while the final fixed vector produced by the Graph Neural Network represents the embeddings for the DeepWukong model. We create four primary embeddings: the SARD dataset (XFGs) for the DeepWukong model (i.e., the scenario in RQ_1), the BigVul dataset (chunks) for the LineVul model (i.e., the scenario in RQ_1), the *Comp-Vul* test dataset (XFGs) for the DeepWukong model (i.e., the scenario in the RQ_2), and the *Comp-Vul* test dataset (chunks) for the LineVul model (i.e., the scenario in the RQ_2). It is worth noting that the embeddings are generated for the test datasets.

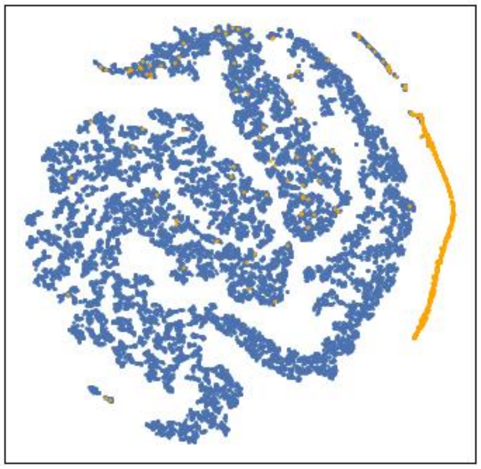
To reduce the dimensionality of the embeddings and plot them in two-dimensional space, we use t-SNE. We create four scatter plots in total, each displaying the reduced embeddings along with their corresponding labels (vulnerable or non-vulnerable). The scatter plots are presented in Figure 6.3.

Results: Both the LineVul and DeepWukong models studied in RQ_1 (refer to Figures 6.3a and 6.3b) exhibit clear separation between vulnerable and non-vulnerable samples of the SARD and Big-Vul datasets, respectively. However, for both the LineVul and DeepWukong models studied in RQ_2 (refer to Figures 6.3c and 6.3d), there is a substantial overlap between vulnerable and non-vulnerable samples of the *Comp-Vul* dataset, which renders it difficult for the models to distinguish between them clearly.

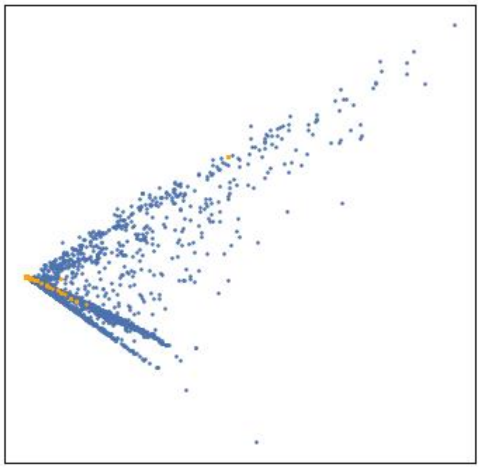
This lack of class separation explains the high number of false positives produced by



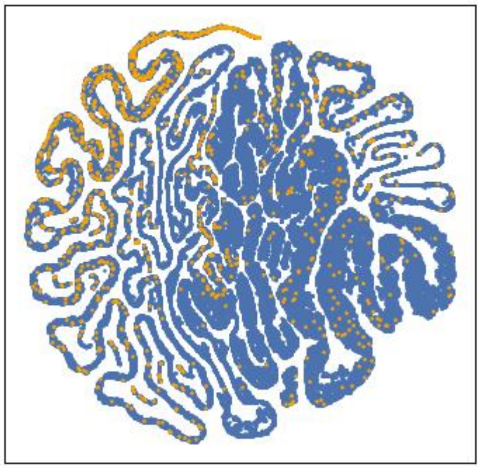
(a) DeepWukong-SARD (RQ_1)



(b) LineVul-Big-Vul (RQ_1)



(c) DeepWukong-*Comp-Vul* (RQ_2)



(d) LineVul-*Comp-Vul* (RQ_2)

Figure 6.3: Scatter plots showing the class separation between the vulnerable and non-vulnerable samples. ● denotes vulnerable samples and ● denotes non-vulnerable samples

the DeepWukong and LineVul models in RQ_2. We can infer that datasets like SARD and Big-Vul are too simple, which allows these models to separate vulnerable and non-vulnerable samples easily. However, when these models are evaluated on a realistic dataset like *Comp-Vul*, they fail to distinguish between vulnerable and non-vulnerable samples.

```
void unix_gc(void)
{
    ....

      spin_lock(&unix_gc_lock)
    ....

      spin_unlock(&unix_gc_lock)


    __skb_queue_purge(&hitlist);

      spin_lock(&unix_gc_lock)
    ....

      spin_unlock(&unix_gc_lock)
}
```

(a) False Positive Function

```
void unix_notinflight(struct
    file *fp)
{
    ....
    spin_lock(&unix_gc_lock)


    BUG_ON(list_empty(&u->link));
    ....

    spin_unlock(&unix_gc_lock)
}
```

(b) Vulnerable Function

Figure 6.4: An example illustrating the pitfall of source code token-based models like LineVul.

6.3 Pitfalls of Token-based Models

The LineVul model [18] is a source-code-token-based vulnerability detection model that takes in a sequence of tokens as input. One of the main problems with the LineVul model is its reliance on lexical relationships between tokens, which can result in the loss of important semantic relationships that are critical for accurately identifying vulnerabilities. Many vulnerabilities in software are caused by complex interactions between different modules or by the programmer's faulty assumptions about how the code will be used. This heavy reliance on source-code tokens to detect vulnerabilities could be one of the reasons for the large number of false positives produced by the LineVul model.

To test the theory that the LineVul model's heavy reliance on lexical relationships between tokens may result in false positives, we conduct an analysis. We collect the functions from the testing *Comp-Vul* dataset that are falsely classified as vulnerable(false positive functions) by the LineVul model in RQ_2. We extract all the vulnerable lines from the vulnerable functions present in the *Comp-Vul* dataset. We then collect false positive functions that contain at least one of the collected vulnerable lines. We randomly selected a few of these functions and analyzed them, one of which is illustrated in Figure 6.4 along with the corresponding matching vulnerable function.

The vulnerable lines that are common between these two functions are highlighted. These functions in the figure are part of the Linux project [1]. The vulnerable function is part of a code in Linux that would allow local users to bypass file-descriptor limits and cause a denial of service attack by sending each descriptor over a UNIX socket before closing it. More information about this vulnerability can be found in the *NVD* website[2]. The highlighted lines in the figure are common to both the vulnerable and false positive functions, but they are not vulnerable in the context of the false positive function. Since the vulnerable lines *spin_lock(&unix_gc_lock)* and *spin_unlock(&unix_gc_lock)* from the vulnerable function *unix_notinflight* are present in the *unix_gc* function, the LineVul model could have incorrectly classified the *unix_gc* function as vulnerable. This analysis supports the theory that models like LineVul, which rely heavily on lexical relationships between tokens, may struggle to model the complex nature of vulnerabilities in real-world software accurately.

[1]https://github.com/torvalds/linux

[2]https://nvd.nist.gov/vuln/detail/CVE-2013-4312

Chapter 7

SliceVul

In this chapter, we explain the motivation behind *SliceVul* and its architectural details alongside the results we obtain when we evaluate *SliceVul* using the *Comp-Vul* dataset.

7.1 Motivation

Despite lacking semantic information about the source code programs, LineVul [18] has been shown to achieve better performance compared to the DeepWukong model [7] when evaluated using the *Comp-Vul* dataset in RQ_2 and RQ_3 experiments. On the other hand, DeepWukong utilizes control and data flow information from the source code programs to detect vulnerabilities. We propose *SliceVul*, a new Deep Learning-based technique that combines the strengths of both these models by utilizing CodeBERT-C [66], a powerful transformer-based model, to learn both the lexical and semantic information of the source code programs. The lexical information is derived from the program slice's source code tokens, while the semantic information is learned from its control and data flow information. *SliceVul* classifies whether a program slice is vulnerable or non-vulnerable.

We compare the performance of *SliceVul* with DeepWukong. By comparing the performance of *SliceVul* and DeepWukong, we aim to answer whether the use of transformer-based models with code semantics is more effective than Graph Neural Network-based models with code semantics in vulnerability detection. Our focus is on program slice-level vulnerability detection models, and therefore, we compare the performance of *SliceVul* and DeepWukong, rather than LineVul.

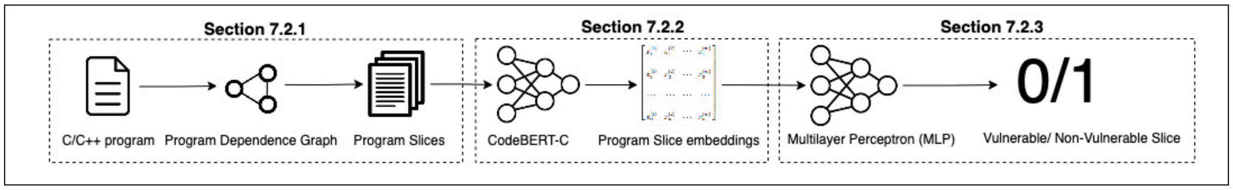


Figure 7.1: An overview of the *SliceVul* architecture

7.2 Model Architecture

Figure 7.1 shows an overview of the *SliceVul* architecture. We explain how we generate the program slices and their embeddings, along with the model training process next in detail.

7.2.1 Generation of Program Slices

SliceVul first computes the Program Dependence Graph(PDG) of the input source code program. A PDG is a directed graph in which each node represents a source code statement, and each edge denotes a data or control flow dependence between two statements. The control dependence between two statements is determined using the control-flow graph (CFG) of a program that captures the execution order of the instructions present in the source code program. The data dependence of a program is determined by using the program's data flow graph that captures the relationship between the places where a variable is assigned and where the assigned value is subsequently used.

Then, to generate the program slices, we conduct forward and backward slicing starting from a program point of interest p_i like method calls, arithmetic operations, pointer expressions, and array indexing. For example, if a program has four method calls and seven arithmetic operation instructions, eleven different program slices are generated for this program. Each of the eleven instructions serves as a separate starting point from which the program slicing is performed. During forward slicing, the program dependence graph (PDG) is traversed forward until a fixed node is reached from p_i, and all visited nodes are added to the forward sliced statements set S_f. During backward slicing, the PDG is traversed backward until a fixed node is reached from p_i, and all visited nodes are added to the backward sliced statements set S_b. The final set of code statements for a program slice is the union of S_f and S_f. The resulting code statements are arranged sequentially in the order they appear in the original source code program to form the input for the vulnerability detection model. A program slice is labeled as vulnerable if it contains at least one vulnerable statement; otherwise, it is marked as non-vulnerable.

7.2.2 Generation of Program Slice Embeddings

We use CodeBERT-C [66] to generate the program slice embeddings. CodeBERT-C is a CodeBERT-based [14] model that is exclusively pre-trained using a huge corpus of C-based projects present in the CodeParrot dataset [1]. The CodeParrot dataset has approximately 14.1M C files with a total size of 183.83GB. We opt for CodeBERT-C instead of CodeBERT as the latter does not include C programs in its pre-training data. By using CodeBERT-C, we can obtain better embeddings for the program slices, as all the projects in our dataset are C/C++-based.

To get the embeddings, we first use the Byte Pair Encoding (BPE) subword tokenization technique [53] to tokenize the source code statements present in the program slices. BPE is an algorithm that separates a long uncommon word into several smaller common subwords. It will aid in minimizing the vocabulary size when tokenizing different custom names rather than immediately adding the whole custom name to the dictionary. For example, the custom name *RelinquishMagickMemory* will be split into a list of subwords, i.e., ['Rel,' 'inqu,' 'ish,' 'Mag,' 'ick,' 'Memory'].

To feed these code tokens into the CodeBERT-C, we convert each token into an embedding vector, a common practice in neural network training. The embedding vector is a combination of a word encoding vector and a positional encoding vector. The word encoding vector is used to represent the meaningful relationship between a given code token and the other code tokens. The positional encoding vector is used to represent the position of a given token in the input sequence. The neural network learns the word encoding vector from scratch during training, while the positional encoding vector is calculated based on the token's position relative to the code sequence. Each code token is now mapped to an embedding vector constructed by concatenating both the word encoding vector and the positional encoding vector of the code token.

The embedding vectors are fed as input to CodeBERT-C, which contains twelve encoder Transformer blocks, each with a bidirectional multi-head self-attention layer and a fully connected neural network. The self-attention layer calculates the attention weight of each code token, producing an attention vector. Bidirectional self-attention allows tokens to attend to both left and right contexts. Attention weights indicate which statements the

[1]https://huggingface.co/datasets/codeparrot/github-code

Transformer model should focus on. Self-attention is used to obtain global dependencies, where weights show how each token is influenced by all others in the sequence. The final layer of CodeBERT-C produces the embedding representation of the program slice input.

7.2.3 Model Training

The output embeddings obtained from CodeBERT-C are used to train a Multilayer Perceptron (MLP) that will classify whether an input program slice is vulnerable or not. The MLP is a feedforward neural network, which means the information flows from the input layer through the hidden layers and then to the output layer without any loops or feedback connections. It contains a series of dropouts and hidden layers. The hidden layers of the MLP contain nonlinear activation functions(*tanh* function) that transform the input into a more complex representation. A dropout layer is a regularization technique used in neural networks to prevent overfitting during training. It randomly drops out (i.e., sets to zero) a fraction of the input units of a layer during each training iteration. This forces the network to learn more robust features and prevents it from relying too much on any one input unit. The last layer of the MLP outputs the probability of a program slice being vulnerable.

The training of the MLP involves adjusting the weights and biases of the neurons to minimize the error between the predicted and actual output. This is accomplished through backpropagation, a technique that computes the gradient of the error with respect to the weights and biases and then adjusts them in the direction that minimizes the error. We use backpropagation with AdamW optimizer [40], which is widely adopted to update the model weights and minimize the loss function. We use the cross-entropy loss function to measure the difference between the predicted and actual labels. The function gives high loss values when the predicted probability is far from the true probability and lower loss values when the predicted and actual values are close. The MLP is trained for multiple epochs to generate a well-trained vulnerability detection model (*SliceVul*). The trained model can then be used for further vulnerability detection.

7.3 Experiment

We evaluate our *SliceVul* model using the *Comp-Vul* test dataset described in section 4.1.3. We use the *Comp-Vul* training dataset in section 4.1.3 to generate the training program slice samples. We use Joern[2] to generate the program dependence graphs. We follow

[2]https://github.com/octopus-platform/joern

the steps described in section 7.2.1 to generate the program slices for both the *Comp-Vul* training and *Comp-Vul* test datasets. After we obtain the program slices, we eliminate the duplicate program slices present in the training *Comp-Vul* dataset. Finally, we obtain a total of 2,961,029 program slices (15,3142 vulnerable program slices + 945,715 non-vulnerable program slices) from the *Comp-Vul* training dataset and a total of 4,853,903 program slices (5,951 vulnerable program slices + 4,847,952 non-vulnerable program slices) from the *Comp-Vul* test dataset. We create a balanced training dataset containing the program slice samples following the same steps explained in section 5.3, which is then used to train the *SliceVul* model. We download the CodeBERT-C model[3] and use it to generate the embeddings for the program slices present in the datasets. We use these embeddings to train the MLP network described in 7.2.3 to classify the program slices as vulnerable or non-vulnerable correctly. We train the model for ten epochs with a batch size of 16. After training, we use the program slice samples from the *Comp-Vul* test dataset to evaluate the trained *SliceVul* model. We use the Hugging Face[4] and PyTorch[5] libraries to train and evaluate the *SliceVul* model.

7.4 Results

Figure 7.2 shows the results of the performance of the *SliceVul* model, trained and evaluated using the *Comp-Vul* dataset. We achieve an accuracy, precision, recall, and F1-score of 99%, 71%, 10%, and 17%, respectively. The accuracy, precision, recall, and F1 scores are up by 10%, 9%, 18%, and 15%, respectively, compared to the RQ_3 DeepWukong model. When comparing these results to the DeepWukong model in RQ_2, we observe an increase of 8%, 9%, and 15% in accuracy, precision, and F1 scores, respectively, while a decrease of 16% in the recall score. Overall, the results suggest that incorporating both syntactic and semantic features of the source code program with a Transformer architecture is more effective than utilizing the semantic properties of the code with a Graph Neural Network. Our study demonstrates that using CodeBERT-C, a pre-trained language model trained on millions of GitHub repositories, specifically on C-based projects, along with the code properties (control and data flow information) of the source code program, is beneficial in achieving high performance for the *SliceVul* model.

While the *SliceVul* model performs better than the DeepWukong model, it still generates a large number of false positives. False positives can lead to a waste of time and

[3]https://huggingface.co/neulab/codebert-c
[4]https://huggingface.co/
[5]https://pytorch.org/

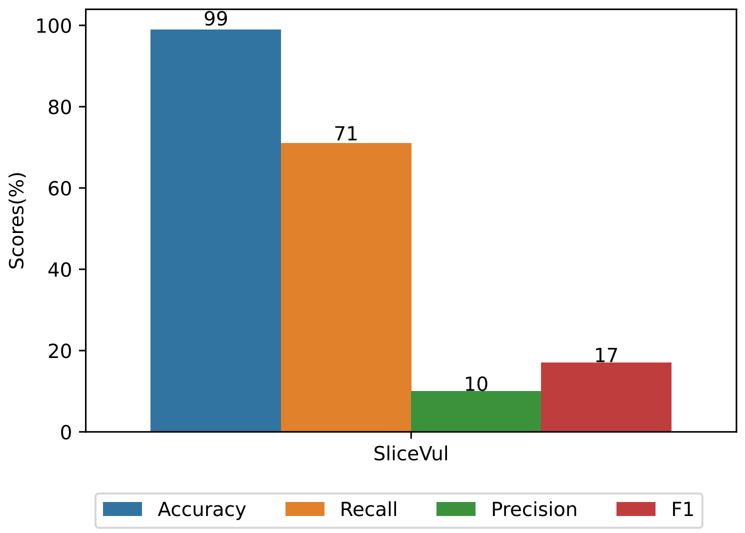


Figure 7.2: Results of the *SliceVul* model when evaluated using the *Comp-Vul* test dataset

resources as they require manual inspection and validation. Moreover, false positives can reduce the confidence of developers in the model and discourage them from using it. Therefore, further research should investigate innovative techniques for reducing false positives in the *SliceVul* model. One potential solution could be to incorporate additional features about the source code that can improve the accuracy of the model. For example, incorporating information about the context in which the code is used, such as the purpose of the function, could help the model make more informed predictions.

Chapter 8

Threats to Validity

Our study uses the Big-Vul dataset [13] to construct *Comp-Vul*. It is possible that some vulnerable samples in the Big-Vul dataset are mislabelled. However, the labelled samples in the dataset were manually verified by Fan et al. [13]. Also, the *Comp-Vul* dataset may not fully represent all real-world scenarios since it is constructed using only ten open-source projects. However, it is worth noting that these ten projects are well-established and popular (e.g., Chrome [1] and Linux [2]) and have been extensively studied and utilized in previous research [67, 3].

In RQ_3, to tackle the class imbalance issue, we propose an undersampling technique where we balance the class samples. Other techniques like oversampling and SMOTE [4] could be used to address the class imbalance issue. Also, in RQ_3, we use balanced datasets where the number of vulnerable and non-vulnerable samples is equal. However, the random selection of samples for the balanced datasets may impact our results since different random selections can lead to different findings [21]. To address this issue, it is recommended to train the models multiple times with various sample sets and examine the outcomes. Unfortunately, this approach was not feasible due to our constrained computational resources.

Another notable concern is the collective training of the DeepWukong and LineVul models on all types of CWE present in the *Comp-Vul* dataset. The complexity and diversity of CWE types could have made it challenging for the models to fully capture the unique characteristics of each individual CWE type. To address this potential limitation, future

[1]https://chromium.googlesource.com/chromium/src/
[2]https://github.com/torvalds/linux

studies could explore the possibility of training separate models for each CWE type and evaluating these models on a dataset like *Comp-Vul*. By adopting this approach, the models could focus exclusively on the specific vulnerabilities associated with each CWE type, potentially improving their detection capabilities.

Furthermore, we focus our work on assessing the efficacy of deep learning-based vulnerability detection models, which can limit the generalizability of our findings to other techniques. Future works should consider evaluating other techniques like static and dynamic analysis tools, traditional machine learning algorithms like SVM, and Random Forest on the *Comp-Vul* dataset and analyze the results. It is important to recognize that potential flaws in our code could have unintended effects on our findings. To minimize this risk, we thoroughly examined and tested our code.

We did not consider the impact of hyperparameters on model performance due to the high computational cost of hyperparameter tuning. Future research should investigate the effect of different hyperparameters on model performance.

Chapter 9

Conclusion and Future Work

In our thesis, we study the performance of deep learning-based vulnerability detection models in realistic vulnerability detection settings. First, we create a new comprehensive, realistic vulnerability detection dataset, called *Comp-Vul*. *Comp-Vul* contains complete source code samples of ten diverse real-world open-source projects. Then, we evaluate two *state-of-the-art* models, LineVul and DeepWukong, on the *Comp-Vul* dataset. Our evaluation indicates a considerable decrease in the model's performance, as evidenced by a drop in precision and F1 scores of up to 95% and 91%, respectively. Furthermore, we find that the ratio of vulnerable to non-vulnerable samples in the evaluation dataset has a considerable impact on the performance metrics of these models. Our investigation also reveals that the embeddings generated by these models depict a substantial overlap between vulnerable and non-vulnerable samples. This suggests that such models struggle to differentiate between vulnerable and non-vulnerable samples in the *Comp-Vul* dataset, resulting in a high number of false positives. We also propose a new Deep Learning-based program slice-level vulnerability technique called *SliceVul* that integrates the strengths of Transformers and the semantic properties(data and control flow information) of source code programs to classify whether a program slice is vulnerable or not. The proposed *SliceVul* model outperforms DeepWukong, a *state-of-the-art* slice-level vulnerability detection model when evaluated using the *Comp-Vul* dataset. Our study argues that when it comes to identifying vulnerabilities in realistic vulnerability detection settings, things may not be as good as they seem, and there is a need for improved model design and evaluation approaches to achieve more accurate vulnerability detection performance.

Future works can focus on creating new techniques that will reduce the number of false positives when evaluated using realistic datasets like *Comp-Vul*. In recent times, large language models(LLMs) like GPT-4[1] produce *state-of-the-art* results in many tasks like classification, recommendation, summarization, and code/text search, etc. They have been trained using a wide range of sources, including books, websites, academic journals, social media platforms, and even GitHub repositories. One possible approach is to use the embeddings of large language models(LLMs) like GPT-4 for source code samples and use these embeddings to build a vulnerability detection model. Another possible solution is to combine the strengths of *state-of-the-art* static analysis vulnerability detection tools with Deep Learning-based techniques that consider the semantics of the source code programs. Another intriguing direction for future research involves training the vulnerability detection models using a dataset where fixed versions of vulnerabilities are treated as non-vulnerable samples. In this approach, the fixed versions would serve as negative examples providing a contrasting perspective that enables the models to learn valuable patterns associated with vulnerabilities. Also, future research is encouraged to explore realistic vulnerability detection datasets from programming languages other than C/C++, such as Python, Java, and JavaScript, in order to broaden our understanding of vulnerability detection in diverse programming contexts.

[1]https://cdn.openai.com/papers/gpt-4.pdf

References

[1] Leo Breiman. 2001. Random forests. *Machine learning* 45 (2001), 5–32.

[2] Cătălina Cangea, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Liò. 2018. Towards sparse hierarchical graph classifiers. *arXiv preprint arXiv:1811.01287* (2018).

[3] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[4] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[5] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[6] Zimin Chen, Steve James Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* (2022).

[7] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–33.

[8] Xiao Cheng, Haoyu Wang, Jiayi Hua, Miao Zhang, Guoai Xu, Li Yi, and Yulei Sui. 2019. Static detection of control-flow-related vulnerabilities using graph embedding. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 41–50.

[9] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

[10] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*. PMLR, 2702–2711.

[11] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2017. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368* (2017).

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[13] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[15] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[16] Yoav Freund, Robert E Schapire, et al. 1996. Experiments with a new boosting algorithm. In *icml*, Vol. 96. Citeseer, 148–156.

[17] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[18] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. (2022).

[19] Daniel Grahn and Junjie Zhang. 2021. An Analysis of C/C++ Datasets for Machine Learning-Assisted Software Vulnerability Detection. In *Proceedings of the Conference on Applied Machine Learning for Information Security, 2021.*

[20] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. 85–96.

[21] Martin T Hagan, Howard B Demuth, and Mark Beale. 1997. *Neural network design.* PWS Publishing Co.

[22] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. 1998. Support vector machines. *IEEE Intelligent Systems and their applications* 13, 4 (1998), 18–28.

[23] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 596–607.

[24] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[25] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: finding unpatched code clones in entire os distributions. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 48–62.

[26] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[27] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 595–614.

[28] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[29] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[30] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. PMLR, 1188–1196.

[31] Yann LeCun, Yoshua Bengio, et al. [n. d.]. Convolutional networks for images, speech, and time series. ([n. d.]).

[32] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. 2017. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 318–328.

[33] Xin Li, Lu Wang, Yang Xin, Yixian Yang, Qifeng Tang, and Yuling Chen. 2021. Automated software vulnerability detection based on hybrid neural network. *Applied Sciences* 11, 7 (2021), 3201.

[34] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[35] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 292–303.

[36] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021).

[37] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[38] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. 2020. Software vulnerability detection using deep neural networks: a survey. *Proc. IEEE* 108, 10 (2020), 1825–1848.

[39] Francesco Lomio, Emanuele Iannone, Andrea De Lucia, Fabio Palomba, and Valentina Lenarduzzi. 2022. Just-in-time software vulnerability detection: Are we there yet? *Journal of Systems and Software* (2022), 111283.

[40] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[41] Stephan Neuhaus and Thomas Zimmermann. 2009. The Beauty and the Beast: Vulnerabilities in Red Hat's Packages.. In *USENIX annual technical conference*. 527–538.

[42] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2022. Generating realistic vulnerabilities via neural code editing: an empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1097–1109.

[43] Marcus Pendleton, Richard Garcia-Lebron, Jin-Hee Cho, and Shouhuai Xu. 2016. A survey on systems security metrics. *ACM Computing Surveys (CSUR)* 49, 4 (2016), 1–35.

[44] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[45] Leif E Peterson. 2009. K-nearest neighbor. *Scholarpedia* 4, 2 (2009), 1883.

[46] Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2010. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 447–456.

[47] J. Ross Quinlan. 1996. Learning decision tree classifiers. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 71–72.

[48] Razvan Raducu, Gonzalo Esteban, Francisco J Rodriguez Lera, and Camino Fernández. 2020. Collecting vulnerable source code from open-source repositories for dataset generation. *Applied Sciences* 10, 4 (2020), 1270.

[49] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[50] Xin Rong. 2014. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738* (2014).

[51] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762.

[52] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering* 40, 10 (2014), 993–1006.

[53] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015).

[54] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: value-flow-based precise code embedding. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

[55] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction.* 265–266.

[56] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[58] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 297–308.

[59] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[60] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* 363–376.

[61] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2017. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference.* 42–54.

[62] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security.* IEEE, 17–26.

[63] Eliezer Yudkowsky et al. 2008. Artificial intelligence as a positive and negative factor in global risk. *Global catastrophic risks* 1, 303 (2008), 184.

[64] Wei Zheng, Jialiang Gao, Xiaoxue Wu, Fengyu Liu, Yuxing Xun, Guoliang Liu, and Xiang Chen. 2020. The impact factors on the performance of machine learning-based vulnerability detection: A comparative study. *Journal of Systems and Software* 168 (2020), 110659.

[65] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: a dataset built for AI-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 111–120.

[66] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. *arXiv preprint arXiv:2302.05527* (2023).

[67] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).