2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00188

An Empirical Study of Deep Learning Models for Vulnerability Detection

Benjamin Steenhoek
Iowa State University
Ames, Iowa, USA
benjis@iastate.edu

Md Mahbubur Rahman
Iowa State University
Ames, Iowa, USA
mdrahman@iastate.edu

Richard Jiles
Iowa State University
Ames, Iowa, USA
rdjiles@iastate.edu

Wei Le
Iowa State University
Ames, Iowa, USA
weile@iastate.edu

***Abstract*—Deep learning (DL) models of code have recently reported great progress for vulnerability detection. In some cases, DL-based models have outperformed static analysis tools. Although many great models have been proposed, we do not yet have a good understanding of these models. This limits the further advancement of model robustness, debugging, and deployment for the vulnerability detection. In this paper, we surveyed and reproduced 9 state-of-the-art (SOTA) deep learning models on 2 widely used vulnerability detection datasets: Devign and MSR. We investigated 6 research questions in three areas, namely *model capabilities*, *training data*, and *model interpretation*. We experimentally demonstrated the variability between different runs of a model and the low agreement among different models' outputs. We investigated models trained for specific types of vulnerabilities compared to a model that is trained on all the vulnerabilities at once. We explored the types of programs DL may consider "hard" to handle. We investigated the relations of training data sizes and training data composition with model performance. Finally, we studied model interpretations and analyzed important features that the models used to make predictions. We believe that our findings can help better understand model results, provide guidance on preparing training data, and improve the robustness of the models. All of our datasets, code, and results are available at https://doi.org/10.6084/m9.figshare.20791240.**

***Index Terms*—deep learning, vulnerability detection, empirical study**

I. Introduction

Deep learning vulnerability detection tools have achieved promising results in recent years. The state-of-the-art (SOTA) models reported 0.9 F1 score [15], [35] and outperformed static analyzers [6], [12]. The results are exciting in that deep learning may bring in transformative changes for software assurance. Thus, industry companies such as IBM, Google and Amazon are very interested and have invested heavily to develop such tools and datasets [19], [27], [30], [46].

Although promising, deep learning vulnerability detection has not yet reached the level of computer vision and natural language processing. Most of our research focuses on trying a new emerging deep learning model and making it work for a dataset like the Devign or MSR dataset [13], [27], [47]. However, we know little about the model itself, e.g., what type of programs the model can/cannot handle well, whether we should build models for each vulnerability type or we should build one model for all vulnerability types, what is a good training dataset, and what information the model has used to make the decisions. Knowing the answers to these questions can help us better develop, debug, and apply the models in practice. But considering the black-box nature of deep learning, these questions are very hard to answer. This paper does not mean to provide a complete solution for these questions but is an exploration towards these goals.

In this paper, we surveyed and reproduced a collection of SOTA deep learning vulnerability detection models, and constructed research questions and studies to understand these models, with the goal of distilling lessons and guidelines for better designing and debugging future models. To the best of our knowledge, this is the first paper that systematically investigated and compared a variety of SOTA deep learning models. In the past, Chakraborty et al. [7] have explored four existing models such as VulDeePecker [24], SySeVR [23] and Devign [47] and pointed out that the models trained with synthetic data reported low accuracies on real-world test set, and the models used spurious features like variable names to make the predictions.

We constructed our research questions and classified them into three areas, namely *model capabilities*, *training data*, and *model interpretation*. Specifically, our first goal is to understand the capabilities of deep learning for handling vulnerability detection problems, especially regarding the following research questions:

- **RQ1** Do models agree on the vulnerability detection results? What are the variabilities across different runs of a model and across different models?
- **RQ2** Are certain types of vulnerabilities easier to detect? Should we build models for each type of vulnerabilities or should we build one model that can detect all the vulnerabilities?
- **RQ3** Are programs with certain code features harder to be predicted correctly by current models, and if so, what are those code features?

Our second study focuses on training data. We aim to understand whether and how the training data size and project composition can affect the model performance. Specifically, we constructed the following research questions:

- **RQ4** Can increasing the dataset size help improve the model performance for vulnerability detection?

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00188

- **RQ5** How does the project composition in the training dataset affect the performance of the models?

Finally, our third investigation area is model intepretation. We used SOTA model explanation tools to investigate:

- **RQ6** What source code information the models used for prediction? Do the models agree on the important features?

To answer the research questions, we surveyed the SOTA deep learning models and successfully reproduced 11 models on their original datasets (see Section II). These models used different deep learning architectures such as GNN, RNN, LSTM, CNN, and Transformers. To compare the models, we managed to make 9 models work with the Devign and MSR, two popular datasets. We selected the two datasets because (1) both of the datasets contain real-world projects and vulnerabilities; (2) the majority of models are evaluated and tuned with the Devign dataset in their papers; and (3) the MSR dataset contains 310 projects and its data have annotations on vulnerability types, which are needed to study our RQs. We discovered the findings for our 6 RQs with carefully designed experiments (Section III) and considerations of the threats (Section IV). In summary, our research contributions include:

1) We conducted a comprehensive survey for the deep learning vulnerability detection models.
2) We delivered a reproduction package, consisting of the trained models and datasets for 11 SOTA deep learning frameworks with various study settings;
3) We designed 6 RQs to understand model capabilities, training data and model interpretation;
4) We constructed the studies and experimentally obtained the results for the RQs; and
5) We prepared interesting examples and data for further studying model interpretability.

II. A Survey of Models and their reproduction

To collect the SOTA deep learning models, we studied the papers from 2018 to 2022 and also used Microsoft's CodeXGLUE leaderboard [1] and IBM's Defect detection D2A leaderboard [2]. We worked with all the open-source models we can find, and successfully reproduced 11 models. The complete list of models and the reasons we failed to reproduce some models are given in our data replication package.

As shown in Table I, the reproduced models cover a variety of deep learning architectures. Devign [47] and ReVeal [7] used GNN on *property graphs* [47] that integrate control flow, data dependencies and AST. ReGVD [31] used GNN on tokens. Code2Vec used *multilayer perceptron (MLP)* on AST. VulDeeLocator [22] and SySeVR [23] are based the sequence models of RNN and Bi-LSTMs. Recent deep learning detection used pre-trained transformers, including CodeBERT [14], VulBERTa-CNN [16], VulBERTa-MLP, PLBART [2] and LineVul [15]

TABLE I: 11 Reproduced Models

Model	Year	Architecture	Dataset
Devign [47]	2019	GNN, property graph	Devign
ReVeal [7]	2021	GNN, property graph	Devign, ReVeal
ReGVD [31]	2022	GNN, token	Devign
CodeBERT [14]	2020	Transformer	Devign
VulBERTa-CNN [16]	2021	Transformer, CNN	VulDeePecker, Draper, ReVeal μVulDeePecker, Devign, D2A
VulBERTa-MLP [16]	2021	Transformer, MLP	
PLBART [2]	2021	Transformer	Devign
LineVul [15]	2022	Transformer	MSR
Code2Vec [3]	2021	MLP, AST	Devign
SeSyVR [23]	2018	RNN	SARD, NVD
VulDeeLocator [22]	2020	Bi-LSTM	SARD, NVD

For our RQs, we used the Devign [47] and MSR [13] datasets. We studied the datasets used in these 11 models in their original papers, shown under *Dataset* in Table I. We found that the Devign dataset has been evaluated and tuned in 8 out of 11 models. It is a *balanced* dataset consisting of roughly the same number of vulnerable and non-vulnerable examples, a total of 27,318 data points (each example is also called an data point). LineVul worked with the MSR dataset, which is a more recently available dataset. It is an *imbalanced dataset*, consisting of 10,900 vulnerable examples and 177,736 non-vulnerable examples. These examples are labeled with their source projects and their Common Weakness Enumeration entries (CWE), which indicates the types of the vulnerabilities. We leverage such traits of the datasets for some of our RQs.

We reproduced the results for the models using their original datasets and settings, shown in Table II. The columns of *A*, *P*, *R* and *F* represents the commonly used metrics in deep learning vulnerability detection, including *accuracy*, *precision*, *recall* and *F1*. Our reproduction results are able to compute within 2% difference compared with the original papers in general. The exceptions are ReVeal, for which the authors confirmed that our results fixed a data leakage error in the original paper, and Devign, for which we used the third-party[3] reproduction released by Chakaborthy et al [7], as the original Devign code is not open-sourced.

To enable the comparisons of the models, we improved the models' implementations to support both Devign and MSR datasets. When running experiments for the RQs, we excluded *VulDeeLocator* and *SeSyVR* as they cannot be easily modified for the Devign and MSR datasets. As a result, we used the rest 9 models for our studies of the RQs.

III. Research Questions and Findings

We organized the research questions into three areas, namely *model capabilities*, *training data*, and *model interpre-*

[1] https://microsoft.github.io/CodeXGLUE
[2] https://ibm.github.io/D2A
[3] https://github.com/saikat107/Devign

TABLE II: Model reproduction on their original datasets. Reproduction results are reported as the mean of 3 random seeds. '-' indicates that the results for a metric were not reported in their papers. The numbers are in percentage.

Model	Paper Results				Our Reproduction			
	A	P	R	F	A’	P’	R’	F’
Devign	59	54	63	57	56	50	71	59
ReVeal	63	57	75	64	53	48	71	56
ReGVD	63	-	-	-	62	62	46	52
CodeBERT	62	-	-	-	64	59	54	55
VulBERTa-CNN	64	-	-	-	64	60	59	59
VulBERTa-MLP	65	-	-	-	63	60	58	59
PLBART	63	-	-	-	62	58	59	59
LineVul	-	97	86	91	99	96	88	92
Code2Vec	62	-	-	-	59	55	58	57
SeSyVR	98	90	92	90	94	88	84	86
VulDeeLocator	99	98	-	97	98	99	96	98

tation. See Sections III-A to III-C respectively. For each RQ, we present the motivation, study setup and our findings.

A. Capabilities of Deep Learning Models

RQ1 Do models agree on the vulnerability detection results? What are the variabilities across different runs of a model and across different models?

Motivation: It is known that deep learning model performance can vary across training runs when using different random seeds. In this RQ, we aim to measure for vulnerability detection, how much such variability actually exists. Additionally, we want to discover how much agreement exists across different deep learning models and across the models with similar architectures. We hope our findings can inform developers and researchers of the uncertainty which potentially exists behind the numbers reported by such tools.

Study Setup: We trained the models using 3 different random seeds on the same train/valid/test partitions of the Devign dataset. We used this dataset because almost all the models tuned their hyperparameters on it. We measured the percentage of *stable* inputs—-an input that has the same binary label for all 3 random seeds. We then compared the stable inputs across the models to measure their agreement.

Findings: In Table III, we reported the percentage of stable inputs for the entire dataset, under *stable-all*, and for the test dataset, under *stable-test*. We also reported the variations of the F1 score across 3 seeds (on test dataset) under *stdev-test-F1*.

Our results show that on average 34.9% test data (30.6% total data) reported different predictions dependent on the seeds used in training. The GNN models that work on property graph ranked the top 2 variability; especially for ReVeal, for 50% of the test data, its outputs changed between runs. Code2Vec reported the least variability compared to the GNN and transformer models. Interestingly, we found that unstable inputs are associated with more incorrect predictions—-stable inputs had a total of 19% incorrect predictions across all seeds and unstable had 47%.

Although many examples reported different predictions between runs, we found that F1 test scores did not change as much, and had a standard deviation of 2.9 on average. That said, for most models, we expect 95% of performance measurements to be within a range of 5.8% above or below the mean performance when measured on multiple random seeds.

TABLE III: Variability over 3 random seeds on Devign dataset

Model	stable-all	stable-test	stdev-test-F1
ReVeal	55%	50%	2.73
Devign	57%	55%	2.24
VulBERTa-MLP	60%	58%	3.13
PLBART	72%	67%	1.03
LineVul	72%	67%	3.46
CodeBERT	72%	69%	2.78
VulBERTa-CNN	74%	71%	2.60
ReGVD	74%	72%	7.33
Code2Vec	89%	77%	0.78

Table IV shows that the deep learning models learned diverse classifiers in that only 7% of the test data (and 7% total data) are agreed by all the models. The 3 GNN models agreed on 20% of test examples (and 25% total), whereas the 3 top performing transformers (LineVul, PLBART, and VulBERTa-CNN) agreed on 34% test data (and 44% total). But when we compared all 5 transformer models, only 22% of test examples (and 29% total) are agreed. The low agreement among different models implies that when there are no ground truth labels, a *differential testing* approach that compares across models as an oracle may have limited uses.

TABLE IV: Agreement across different models

Model	agreed-all	agreed-test
All 9 models	7%	7%
All 3 GNN models	25%	20%
Top 3 transformer models	44%	34%
All 5 transformer models	29%	22%

RQ2 Are certain types of vulnerabilities easier to detect? Should we build models for each type of vulnerabilities or should we build one model that can detect all vulnerabilities?

Motivation: In traditional software assurance techniques such as program analysis, we use different algorithms to detect different vulnerabilities. Certain types, e.g., infinite loops, are more difficult to detect than other types, e.g., memory leaks, because one requires to track symbolic values and reasoning about the loops, and the other only needs to check if the memory free is invoked after its allocation. In this RQ, we are interested to learn whether for deep learning vulnerability detectors, it is also true that certain types of vulnerabilities are easier to detect than the others. Considering different types of vulnerabilities have different semantics and root causes, we also want to gain some insights as to whether we should build a model for each type of vulnerability or for vulnerability in general (without separating them into the types), like most current work has done.

Study Setup: Here, we study models based on the vulnerability types, and thus we used the MSR dataset. The examples in the MSR dataset are annotated with CWE [4]. Using these CWE types, we group the vulnerabilities into 5 categories, namely *buffer overflow*, *value error*, *resource error*, *input validation error*, and *privilege escalation*, shown under *Vulnerability Types* in Table V. Our criteria are that (1) each group contains the bugs of similar root causes and semantics, and (2) each group has a sufficiently large dataset for effectively training the models. Column *Total* lists the number of the examples collected from the MSR datasets, including all vulnerable and their patched examples that have a CWE annotation.

Specifically, buffer overflow is caused by reading or writing to the memory outside the bounds of the buffer, e.g, CWE-125 “Out-of-bounds Read” and CWE-787 “Out-of-bounds Write”. The mapping of the complete CWE list to the 5 groups is given in our dataset. *Value error* includes the examples of CWE-190 “Integer Overflow”, CWE-369 “Divide By Zero” and CWE-682 “Incorrect calculation”. Such errors are caused by propagating incorrect values through data processing or arithmetic operations. *Resource error* is caused by incorrectly freeing or using a resource such as memory or a file pointer, and are typically detected using typestate analysis [36]. The CWE examples include CWE-415 “Double Free” and CWE-404 “Improper Resource Shutdown”. *Input validation error* is caused by using an external input without validating whether it is correct/benign, e.g., CWE-134 “Use of Externally-Controlled Format String” and CWE-89 “Improper Neutralization of Special Elements used in an SQL Command” (‘SQL Injection’). They are often detected using taint analysis [40]. Finally, *Privilege escalation* is caused by missing proper permission checks and allowing an unauthorized entity to execute privileged commands or view privileged data, such as CWE-264: “Permissions, Privileges, and Access Controls”, and CWE-255: “Credentials Management Errors”.

TABLE V: Five Types of Vulnerabilities

Vulnerability Type	Total	CWE examples
Buffer overflow	37,291	CWE-125, CWE-787
Value error	15,126	CWE-190, CWE-369
Resource error	33,748	CWE-415, CWE-404
Input validation error	25,514	CWE-134, CWE-89
Privilege escalation	32,749	CWE-264, CWE-255

We partitioned the dataset for each bug type into train, valid, and test datasets with 80%/10%/10% ratios. We trained 5 models using the 5 groups of bugs respectively, as well as a *Combined Model* trained with all the bug types for comparison. This reflects the real-world scenarios in that a model trained with specific vulnerability type may be more focused, but the combined model can train with more data. We report the *same-bugtype* performance and *cross-bugtype* performances for each model. The same-bugtype performance reports the test F1 score when the training and test data have the same bug type. The cross-bugtype performances report the test F1 scores when the training and test data have different bug types.

In this experiment, VulBERTa-CNN, VulBERTa-MLP and some models for Code2Vec did not report valid results because they always predicted the same class on the test data.

Findings: We present out results in Figure 1. The bars report the F1 score for the same-bugtype setting, and the circles report the cross-bugtype performances. Each bug type is associated with 4 cross-bugtype test sets, and thus we have four circles for each bar, except that the combined model has five circles, each of which represents running tests for a bug type on the combined models.

Analyzing same-bugtype performance, we found that the models did not always agree on which type of vulnerability is the easiest, and different types of vulnerabilities have achieved the best F1 score in different models. Interestingly, Input validation and Resource errors (the orange and red bars) often reported lower performance than the other types. On the contrary, Buffer overflows and Value errors (the blue and purple bars) often reported better performance compared to other types. In traditional program analysis, these types are harder to detect because they require tracking variable values, and sometimes reasoning about loops.

Devign and ReVeal used GNN architectures on the property graphs. Their bars show the similarities. For the rest of the models, resource errors (the red bars) manifest the lowest performance among the 5 vulnerability types. One possibility is that resource allocation and free can be located far apart in the code, and the transformer models cannot capture such long range dependencies. Another possibility is that such errors cover a variety of resources, and the training dataset may not contain sufficient data for each resource for the model to extract the patterns.

The combined model (the brown bar) is generally less performant compared to the models trained with a specific type of vulnerability, but for some vulnerability types like Input validation and Resource errors, the combined model can perform better. For example, for CodeBERT, the combined model achieved higher F1 compared to all of the other 5 models. The circles inside the brown bar show that Privilege escalation and Resource errors reported relatively low accuracy, but for all of the vulnerability types, the combined model reported better performance compared to the type specific models.

Analyzing cross-bugtype performance, we found that for the most of time, cross-bugtype detection reports much lower performance except LineVul, implying that different vulnerability types represent different data distributions. LineVul seems to handle value errors very well. When applying models

[4] https://cwe.mitre.org

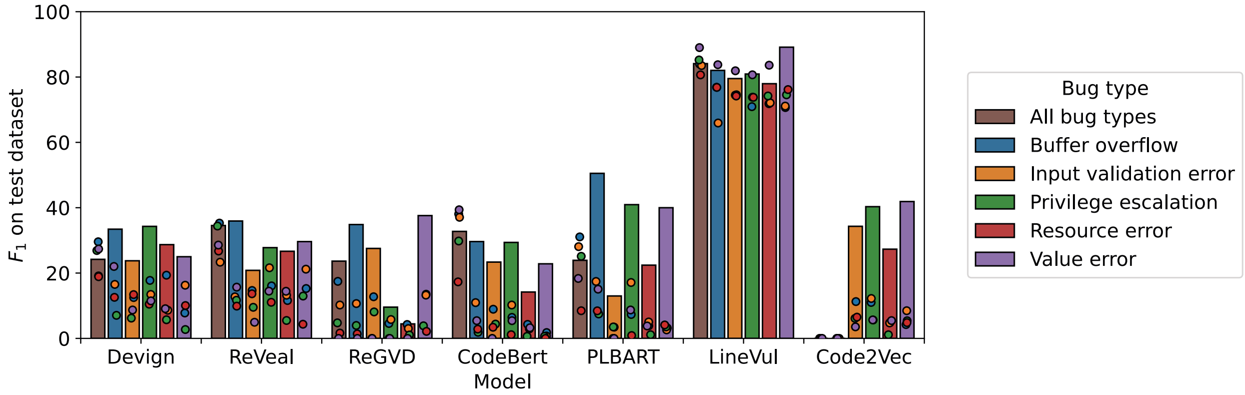


Fig. 1: Same-bugtype and cross-bugtype performance. Bars indicates same-bugtype performance; circles indicate cross-bugtype performance for each other bugtype.

trained with other vulnerabilities, value errors reported higher performance than the same-bug performance.

RQ3 Are programs with certain code features harder to be predicted correctly by the current vulnerability detection models, and if so, what are those code features?

Motivation: Here, we investigated whether we can characterize the programs that cannot be predicted well and are "hard" to deep learning models, and whether different models agree on such difficulties. Knowing what programs we cannot handle gives a good target for our future work to improve upon. In program analysis, we know that certain features are hard to handle, such as loops and pointers. We want to know whether these features are also hard for deep learning.

Study Setup: In the first step, we prepared a list of code features for investigation. We think it is interesting to compare with program analysis tools regarding what types of programs are hard to handle. So our approach is to list code features that are important to program analysis, and then check if they also made a difference for deep learning tools.

We obtained a total of 12 code features. Some are control flow related, e.g., the structures of *while*, *for*, *if*, *goto*, *call* and *switch* as well as *unconditional jumps* of break, continue, return; some are data structure and pointer related, e.g., *arrays* and *pointers* (including the field accesses); and finally some are auxiliary structures such as *comment* and *macro*. Based on this feature list, we applied the *tree-sitter* [5] parser to count the frequency of code features present in each function.

To understand whether certain code features make deep learning models harder to predict, we used a *multivariate logistic regression (LR)* model (see Eq. 1) to associate the code features with the likeliness of a function being predicted correctly. If a function with certain code features is more likely predicted correctly, we consider it as easier to deep learning, and vice versa. Given a function with a particular feature composition, Y in Eq. 1 reports the predicted probability that the deep learning model will predict it correctly. x_i is the count of each code feature in the function. β_is are the coefficients learned from the data. Each β_i is associated with one code feature x_i. When β_i is negative, the value $\beta_i * x_i$ decreases the predicted probability of correctness, so we term these features to be *difficult* for the model. Likewise, code features with positive coefficients are termed to be *easy*. Meanwhile, a large β_i implies that the increase in the count of code feature x_i greatly increases the predicted probability of correct prediction, and vice versa.

$$Y = \sigma(\sum_i \beta_i * x_i + \beta_0) \tag{1}$$

We trained the LR model on the predictions made on the validation set, and then used the trained LR model to find difficult/easy examples from the test set. To quantify the difficulty of an example in test set, we used the logit input to the sigmoid function in the LR model: $\ell(x) = \sum_i \beta_i * x_i$. We denote the negation of this quantity as the *difficulty score*. A function with a higher difficulty score is expected to be more likely predicted incorrectly than a function with a lower difficulty score. To evaluate the effectiveness of this LR model, we selected the top and bottom 10% of examples in the test set, sorted by their difficulty scores. We then evaluate the predictions of our LR model by comparing the model performance on the easy and difficult datasets we selected.

It should be noted that initially we have tried statistical significance tests and correlation-based methods to link code features with the model accuracy. Then, we realized that these approaches consider only one feature a time. For example, the functions with 3 loops, 400 pointers, and 500 macros perform well, while those with 300 loops, 300 pointers and 100 macros perform bad. We cannot conclude whether the performance difference is due to the loops, pointers or macros. On the other hand, the LR model incorporates multiple code features at once and considers the effect of combinations of the features.

[5] https://tree-sitter.github.io/tree-sitter/

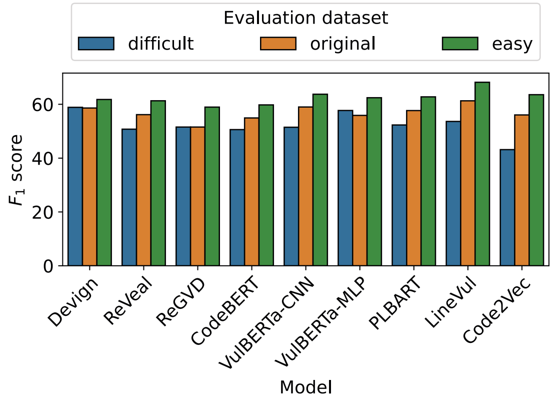


Fig. 2: Comparative performance on evaluation sets selected according to LR model difficulty score, averaged over 3 random seeds on the Devign dataset. “Original” is the performance on the original test set, reported in Table II.

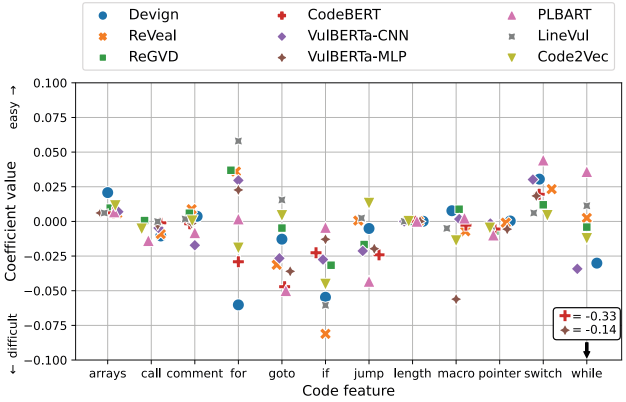


Fig. 3: Coefficients of LR models trained on the stable examples from the Devign dataset.

Findings: Figure 2 shows that all 9 models performed better on the easy dataset than on the difficult set. The average difference between easy/difficult performance was 10.3% for all models. For the majority of models (7 out of 9), the original test set performance lies between the performances of difficult and easy sets. These results demonstrate that the LR model and difficulty score are effective for choosing difficult and easy examples for the deep learning models.

Figure 3 plots the coefficients of each code feature in the LR model trained for each deep learning model. We found that the dots for the features `call`, `length`, and `pointers` are grouped together for all the models, implying that all the models agreed on the importance of these features. Interestingly, all of these dots are located near 0, which indicates that the features did not have a large effect. On the other hand, the features `for`, `goto`, `if`, `jump`, `switch`, and `while` varied the across models. These are all control flow related structures.

We also observed that for all the models, `arrays` and `switch` are associated with the positive coefficients, and that for the majority of models, `if`, `goto`, and `while` fall into the negative ranges. In particular, `if` had a negative coefficient for all the models. The high number of `if`, `goto`, and `while` in a program indicates its high cyclomatic complexity [29], and this type of program is also challenging for program analysis. Especially, property graph-based models like Devign use control flow information, so it makes sense that features `for`, `goto`, `jump` and `while` were all negative (hard).

B. The Training Data

RQ4 Can increasing the dataset size help improve the model performance for vulnerability detection?

Motivation: High-quality vulnerability detection data are hard to obtain. In the past, we used automatic labeling methods such as static analysis and mining change commits. These approaches can introduce incorrect labels [7]. We also used manual labels [47], which are slow to produce [25]. This research question helps us understand whether currently available datasets are large enough to train the models, and whether increasing the dataset size can significantly improve the model performance.

Study Setup: To investigate this RQ, we combined the Devign and MSR datasets, namely, *Imbalanced-dataset*. Since the projects used in Devign overlap with the projects used in MSR, we excluded 82 duplicated examples where the commit IDs are matched. This generates a dataset of 194,285 examples in total. Some of the published models are originally tuned on the balanced model such as Devign, so we also constructed a *Balanced-dataset* of 45,363 examples by taking all the vulnerable examples in MSR and then randomly undersampling an equal number of non-vulnerable examples. For each dataset, we held out 10% data as the test set for all the models. Then we prepared 10%, 20% ... 90%, 100% of the rest of the data to train 10 models to observe how the F1 score for test set changes when the dataset size increases. In addition, we prepared two small datasets of 1% and 5% of the total data to experiment what is the minimum amount of the data needed for the model being able to learning something.

Findings: We summarized our results in Figure 4. Figures 4a and 4b show a similar trend. Generally, all the models increased in performance when we added more data. However, the improvement was not significant. Comparing 100% data with 10%, the F1 on test set reported no difference when we take an average over all the models for the Balanced dataset. For the Imbalanced dataset, the value of F1 score improved 0.16 on average.

In Figure 4a, among the models, only LineVul showed consistent improvement when we added 10% more data each time. All other models fluctuated when we increased the training dataset, which indicates that the increasing data does not always bring benefit. For example, for VulBERTa-MLP, the F1 value drops an average 0.1 over the course of increasing the dataset sizes. The performance of the model trained with 100%

of the dataset is 0.08 less than the performance of the model trained with 10% of the dataset. It seems that other factors rather than the dataset size has played a much significant role in terms of performance. In Figures 4b, ReVeal is the model that improves the most with the increased dataset. At 100% dataset, ReVeal is catching up the best model LineVul. However, Devign, which has a similar architecture of using GNN on the property graph, does not show this benefit of additional data.

The experiments on models with the small datasets (consisting of 1% and 5% total data) showed that surprisingly, we can bring the models up to good performance only using 5% (about 2268 data points and 1134 vulnerable examples) for most of the models except CodeBERT, when learning with the balanced data. When learning with the imbalanced data, the turning point comes a little later. For example, ReGVD and CodeBERT require 50% (96.4 k) and 30% (57.8 k) of the total data. The other models need about 5–10% of the data (9.7–19.4 k) to reach a high point. Interestingly, this dataset has 10.8% vulnerable examples; that said, the models needed about 1048-2095 vulnerable examples to achieve good performance—a comparable amount of vulnerable examples to the setting of the Balanced dataset.

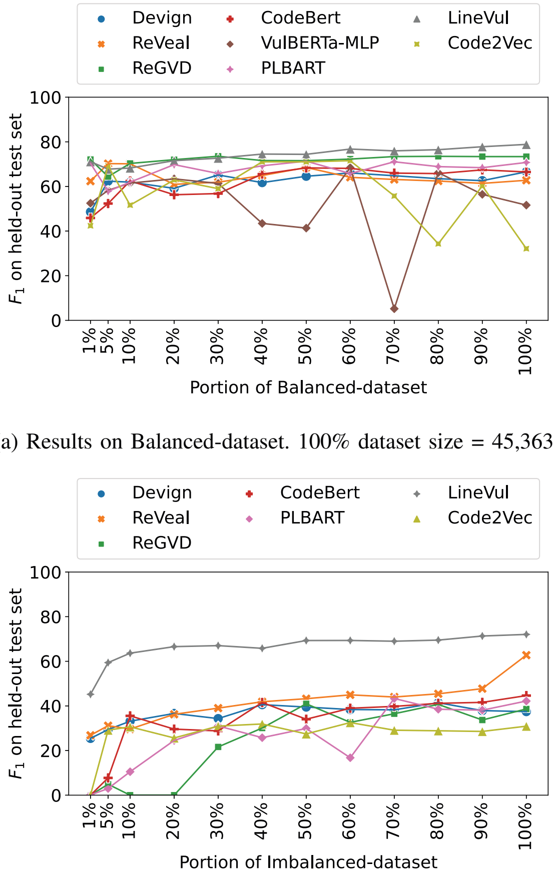
RQ5 How does the project composition in the training dataset affect the performance of vulnerability detection models?

Motivation: In this RQ, we aim to further understand how to compose a good training dataset for vulnerability detection. Specifically, we are interested to know whether the diversity of the projects in the training dataset helps. We are also interested to learn whether different projects indeed represent different distributions such that when the test and training data come from the same projects, the model can significantly perform better, and when the training and test data are from different projects, whether the models can generalize over unseen projects.

Study Setup: We designed two experiments for this study. In the first experiment, we prepared a *non-diverse* training dataset and a *diverse* training dataset, and compared the models trained with the two datasets on the same test set. In the MSR dataset, we found that *Chrome* contains 76k examples and is the largest among all the 310 projects. We used it as the non-diverse dataset. We performed this experiment in a 5-fold cross validation setting to eliminate the potential biases that may exist when selecting projects. For each fold, we randomly sampled 10 k examples from the MSR dataset as a test set. We then excluded Chrome and the projects used in the test set, and randomly sampled a total of 76 k examples (the same number as Chrome has) from the remaining projects. The average number of projects in the diverse dataset is 50.6 across 5 folds.

In the second experiment, we prepared a *mixed-projects* setting where the test set is separated from the training set without considering the source project, and some examples in the training and test sets may originate from the same projects. This is the setting where most of our deep learning papers are evaluated with. We also constructed a *cross-projects* setting where the test set examples must originate from different projects than the projects represented the training set. This setting helps us understand whether we will have a significant performance degradation when using an off-the-shelf trained deep learning vulnerability detection models that have not seen the test projects.

We also used the MSR dataset in a 5-fold cross validation setting. For each fold, we first constructed a test set for the cross-project setting by including all the examples from randomly chosen projects, until the set contained at least 10k examples. Because each project had a different number of examples, the resulting set was slightly larger than 10k examples. We then constructed a test set for the mixed-project setting by randomly partitioning the remaining examples into test (10 k), validation (10 k), and training (the remaining examples, about 158 k) sets. We trained the model, using the 158 k training examples and 10 k validation set, then ran it on the test sets for both the cross-project setting and the mixed-



(a) Results on Balanced-dataset. 100% dataset size = 45,363.

(b) Results on Imbalanced-dataset. 100% dataset size = 194,285.

Fig. 4: F1 score on a held-out test set when models are trained with increased portions of the training dataset.

project setting.

Findings: The results for the first experiment are presented in Figure 5. We used the boxplot to summarize the results of 5-fold cross-validation. To our surprise, we found that for all the models, the diverse training set does not provide any benefits compared to the training set that only consists of *Chrome*. In fact, 5 out of 6 models reported a higher median performance when trained on non-diverse data than on diverse data.

For the second experiment, Figure 5b shows that mixed projects perform significantly better than cross projects for all the models (the average and the largest differences in F1 score were 0.11 and 0.32 respectively). This implies that seeing the data from one project can indeed help predict other data from the same project. Vulnerability detection can greatly benefit from customized trained models compared to directly used already trained off-the-shelf models. For LineVul, the 5 folds reported very different performances in the cross-project setting, indicating that given a target test set, some projects are more useful for training than others. The results also imply that the models may learn to detect vulnerabilities from project-specific attributes of the dataset, such as style, language features or naming conventions. We believe that this motivates further research into causal detection of bugs for generalization.

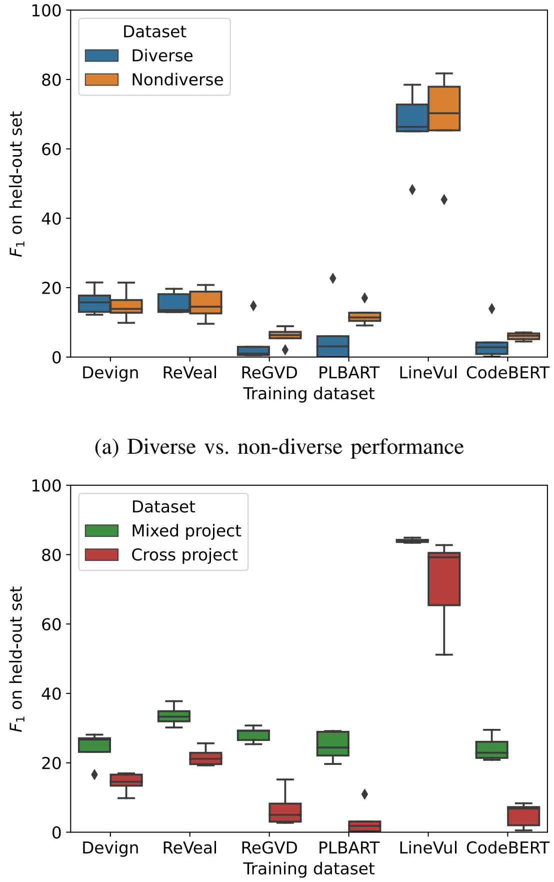
C. Internals of Deep Learning

RQ6 What code information do the models use for prediction? Do the models agree on the important features?

Motivation: Recent deep learning vulnerability detectors have achieved high performance, e.g., LineVul reported 91% F1 score. We want to know why these tools can perform well, and whether the model has used any aspects of semantics of the vulnerability to make decisions. For example, to detect buffer overflows, a semantic-based program analysis tool identifies the dependent statements and reasons about the string lengths and buffer sizes. We also want to investigate whether different models agree on what are the important features.

Study Setup: We surveyed a set of SOTA deep learning interpretation tools, especially for GNN and transformer architectures. We used GNNExplainer [44] for Devign and ReGVD, and LIT [33], [37], [39] for LineVul, VulBERTa-CNN, VulBERTa-MLP, CodeBert and PLBART, as among all the tools we investigated, these two tools can work with the most for our models. In our data reproduction package, we documented the reasons why the rest of the models cannot work with GNNExplainer and LIT, as well as the other intepretability tools we have tried.

To explain the models, both of GNNExplainer and LIT provide scores to measure the importance of the code features. GNNExplainer gives a score for each edge in the graph, and LIT gives a score for each token in the program. To compare the results reported by GNNExplainer and LIT, we performed the following *normalization* on the output of the tools. For GNNExplainer, we calculated the score of each node by taking the average scores of all of its incident edges, as what has been done in [21]. Each token in the node will use this score. For both GNNExplainer and LIT, we calculated the score for each source code line by summing up the scores for all the tokens of that line, following the literature [15]. For each example in the test dataset, we selected the top 10 highest scored lines, as done in [15], [21], to form the *important feature set*, denoted as *I*. We considered these 10 lines as the most important code features the model used to make a decision.



(a) Diverse vs. non-diverse performance

(b) Mixed-project vs. cross-project performance

Fig. 5: Studies on project composition in training data. The bar shows mean F1 and the interval shows standard deviation.

To measure the similarity of the two important feature sets I_A and I_B reported by the models A and B, we calculated the intersection $I_{AB} = I_A \cap I_B$. We also used *Jaccard index* [20] as another metric, defined as:

$$J(I_A, I_B) = \frac{|I_A \cap I_B|}{|I_A \cup I_B|} \tag{2}$$

To report the similarity, we first compute I_{AB} and $J(I_A, I_B)$ for each program in the test set, and then we took the average for I_{AB} and $J(I_A, I_B)$ respectively.

We sampled examples from the following groups for manual inspection: 1) the example is vulnerable, and the model correctly detected it (correct); 2) the example is non-vulnerable, and the model predicted as vulnerable (false positive); and 3) the example is vulnerable, and the model predicted as non-vulnerable (false negative).

Findings: In Table VI, we reported the similarity of the important feature sets from every model pair. Our results show that Linevul and ReGVD have a maximum overlap among all the model pairs. Among the 10 lines ranked as the important features, the two models shared an average of 6.88 lines. We found it interesting that although the models can have a lot of disagreement on individual predictions (See Table IV), the code information they used overlap. All the model pairs have at least 3 lines in common for the important features. Devign, as the only GNN model based on the property graph, has a low overlap with the other models and the lowest is with PLBART, on average 3.38 lines. PLBART used a different transformer architecture compared to the other transformer model, and also reported low overlaps with other transformer models.

We have also analyzed the corrected predicted examples using the same approach as Table VI. We found that the important feature sets have more overlaps for the corrected examples, e.g., Linevul and ReGVD still have the most in common, sharing 7.29 lines in the important feature sets.

TABLE VI: The similarity of important feature sets between every two models measured by I_{AB} (reported in blue) and $J(I_A, I_B)$ (in black). The max and min values are bold.

Model	L-Vul	CodeB	PLBART	Devign	ReGVD	V-CNN	V-MLP
L-Vul	-	0.58	0.31	0.38	**0.60**	0.32	0.40
CodeB	6.84	-	0.37	0.35	0.50	0.31	0.40
PLBART	4.08	4.89	-	**0.24**	0.27	0.30	0.28
Devign	4.90	4.54	**3.38**	-	0.42	0.27	0.30
ReGVD	**6.88**	5.86	3.58	5.36	-	0.33	0.40
V-CNN	4.20	4.06	3.88	3.87	4.35	-	0.30
V-MLP	4.83	4.74	3.71	3.95	4.83	3.97	-

From our manual inspection, we observed that the models commonly highlighted code lines of `for`, `if`, and `while`, as well as the function signatures as important features. The models also often highlighted memory operations of `alloc`, `memset`, and `memcpy`, as well as the lines which print error messages containing `error` or `printf`. To confirm this observation, we performed profiling on the code using these keywords and reported the results in Table VII. Using `error` as an example, without loss of generality, the first 7 rows report the probability of `error` occurring in the important feature set for each model (total number of `error` in the important feature set/total number of lines in the important feature set). In Row *Func*, we show the probability of `error` occurring in a function (total number of `error`/total number of lines in a function). Comparing the two, we show that the probability of `error` occurring in the important feature sets is 2.79 times of the probability of `error` occurring in the program on average, shown in Row *I/F*. This implies that `error` is preferred to be selected into the important feature sets. Among all the features, `error`, `print`, and `alloc` ranked the highest ratios.

TABLE VII: The frequently highlighted code features

Model	error	print	alloc	for	memset	memcpy	while	if
L-Vul	.035	.021	.017	.042	.003	.003	.005	.115
CodeB	.027	.015	.015	.032	.004	.002	.004	.111
PLBART	.025	.012	.011	.034	.003	.001	.005	.114
Devign	.023	.012	.014	.069	.002	.002	.007	.147
ReGVD	.034	.027	.022	.038	.004	.005	.005	.132
V-CNN	.024	.014	.011	.028	.002	.002	.004	.116
V-MLP	.031	.018	.015	.020	.003	.003	.004	.095
Func	.010	.010	.010	.026	.002	.002	.004	.108
I/F	2.79	1.78	1.52	1.47	1.39	1.21	1.12	1.10

```
1  static int asf_read_ext_content(AVFormatContext *s,
   ↪ const GUIDParseTable *g) // (3)
2  {
3      ASFContext *asf  = s->priv_data;
4      AVIOContext *pb  = s->pb;
5      uint64_t size    = avio_rl64(pb); //(7)
6      uint16_t nb_desc = avio_rl16(pb); //(8)
7      int i, ret;
8      for (i = 0; i < nb_desc; i++) {
9          uint16_t name_len, type, val_len; //(5)
10         uint8_t *name = NULL;
11         name_len = avio_rl16(pb);
12         if (!name_len)
13             return AVERROR_INVALIDDATA; //(6)
14         name = av_malloc(name_len);
15         if (!name)
16             return AVERROR(ENOMEM); //(9)
17         avio_get_str16le(pb, name_len, name, //(4)
18                          name_len); //(2)
19         type     = avio_rl16(pb); //(10)
20         val_len = avio_rl16(pb);
21         if ((ret = process_metadata(s, name,
           ↪ name_len, val_len, type, &s->metadata))
           ↪ < 0) //(1)
22         ret = process_metadata(s, name, name_len,
           ↪ val_len, type, &s->metadata);
23         av_freep(&name);
24         if (ret < 0)
25             return ret;
26     }
27 }
```

Listing 1: Memory leak successfully detected by LineVul. The top-10 lines reported by LIT are highlighted in yellow.

Our second observation is that the transformer models sometimes made predictions without seeing the root cause. This is because the transformer models take a fixed-size input, and some code, sometimes including the root cause, is truncated. Interestingly, those models are still able to correctly predict whether a function is vulnerable with high F1 score.

Third, we inspected the vulnerabilities that all the models missed and studied the important feature sets used to detect such vulnerabilities. We found that these vulnerabilities are very application specific. The bugs are missed may because there are not sufficient training data for such types of bugs.

In Listing 1, we show an example where the prediction is correct, but the features used are not causal. The example contains a memory leak vulnerability, and LineVul predicted the example as vulnerable. The memory allocated to `name` at line 14 is never released. The patch at line 23 showed a fix. The important feature set (top-10 lines) reported by LIT are

```
1  static int decode_frame(AVCodecContext *avctx,
2                              void *data, int *got_frame,
                               ↪  AVPacket *avpkt) // (1)
3  {
4      //...10 lines
5      bytestream2_init(&s->gb, avpkt->data,
       ↪  avpkt->size); // (5)
6      if ((ret = ff_tdecode_header(&s->gb, &le,
       ↪  &off))) { // (6)
7          av_log(avctx, AV_LOG_ERROR, "Invalid TIFF
           ↪  header\n"); // (3)
8          return ret;
9      } else if (off >= UINT_MAX - 14 || avpkt->size <
       ↪  off + 14) { // (4)
10         av_log(avctx, AV_LOG_ERROR, "IFD offset is
           ↪  greater than image size\n"); // (2)
11         return AVERROR_INVALIDDATA;
12     }
13     s->le          = le;
14  // TIFF_BPP is not a required tag and defaults to
    ↪  1 // (10)
15     s->bppcount    = s->bpp = 1; // (9)
16     s->photometric = TIFF_PHOTOMETRIC_NONE; // (7)
17     s->compr       = TIFF_RAW; // (8)
18     // ...140 lines
```

Listing 2: Non-vulnerable code is predicted as vulnerable because of spurious features.

highlighted in yellow. We can see that it includes the "patterns" we have discussed, including function signature at line 1, the lines that contain `ERROR`, e.g., lines 13 and 16, as well as the `if` statement at line 21. We also see that for this project, variable `name_len` is important and included multiple times. However, none of the 10 lines cover the memory allocation at line 14, which is important to understand this bug.

This example indicates that the models try to capture patterns of a vulnerability, instead of reasoning about the values, and have difficulty capturing long range semantic dependencies in the code. But we also observed in other examples that sometimes, the control structures and memory statements highlighted as important (see Table VII) can be a part of the dependent statements of the vulnerability, and thus they are useful for inspecting the root cause of the bugs.

Listing 2 shows an example of a non-vulnerable function which LineVul erroneously predicted as vulnerable. The model highlighted the function signature (line 2), lines with "ERROR" (lines 7 and 10), initialization routines (line 5), `if` (lines 6 and 9), and field assignments (lines 15-17), and predicted the function as vulnerable. It showed that making decisions based on the patterns of these structures can lead to mistakes.

IV. Threats to Validity

Our observations are drawn from the models and data sets available and may not be generalized for deep learning vulnerability detection in general. We used both a balanced dataset (Devign) and an imbalanced dataset (MSR) to mitigate this threat. The two datasets both included real-world bugs. Devign is used by most of the models in their evaluation, so we need it to reproduce the models (see Table I). However, our datasets may still not be representative of the real-world vulnerability distribution. We included all the models that we could find and reproduce.

The grouping in RQ2 is subject to bias in that different researchers may divide vulnerability types differently. Here, two of the authors who have the domain knowledge inspected the CWE list individually and discussed and agreed on the grouping. To mitigate the bias that may be brought in by the specific project compositions, RQ5 performed 5-fold cross-validation. For RQ6, we selected the SOTA model interpretation tools; however, such techniques may not be perfect to identify the important features that the models use. The experiments for RQ2, RQ4 and RQ5 require the models to work with our customized data that are not shipped with the models. We tried different random seeds for any suspicious data we have observed, e.g., when a model reported all 0s or 1s. We excluded such models in our results when tuning could not resolve the issues.

V. Related Work

Several works have done empirical studies of machine learning based vulnerability detection models. Chakaborthy et al. [7] studied 4 DL models, and investigated the issues of synthetic datasets, data duplication and data imbalance, and pointed out the use of spurious features, then used these to improve their model design. Tang et al. [38] aim to determine which neural network architectures, vector representation methods, symbolization methods are the best. They surveyed 2 models. Mazuera-Rozo et al. [28] evaluated 1 shallow and 2 deep models on binary classification and bug type (non-binary) classification. After we completed our study, we found two related empirical studies. Lin et al. [26] evaluated 6 DL models' generalization for 9 software projects. Ban et al. [4] evaluated 6 machine learning models (1 of which is a neural network) in a cross-project setting with 3 software projects, and also studied training on 2 bug types vs. a single bug type.

Recently, many vulnerability detection models are proposed with a variety of architectures, such as MLP [9], RNN [22]–[24], [45], CNN [34], [42], Transformer [2], [11], [12], [14]–[16], [32], [41], and GNN [5]–[8], [10], [17], [18], [21], [31], [35], [43], [47]. For example, Devign used gated graph neural network on property graphs [1]. LineVul [15] used a transformer model pretrained over a large body of diverse open-source projects. ReVeal [7] applied SMOTE to address the data imbalance issue and triplet loss to learn to maximally separate vulnerable and non-vulnerable code.

In these papers, most models were evaluated on in-distribution data, where the training set can contain projects and bug types which overlap with the test set. Russell et al. [34], Li et al. [24], and Xu et al. [43] trained their models to detect specific kinds of vulnerabilities, and all found that some vulnerabilities were more difficult than others. Hin et al. [18] evaluated their model in a cross-project setting by holding out one project at a time and found that the performance was slightly degraded. Most model evaluations compared different baselines on metrics such as F1, but did not quantify the agreement on the predictions. To the best of our knowledge, our work is the first attempt to characterize the programs and code features which the model cannot predict well.

VI. Conclusions and Future Work

To understand deep learning vulnerability detection models, we performed an empirical study with 6 research questions. We experimentally show that on average, 34.9% test data have different predictions between runs, and only 7% of predictions are agreed across 9 models. Vulnerability detection based on a specific type generally performs better than a model built for all vulnerabilities. The model performance does not increase significantly with an increased dataset, and for both balanced and imbalanced datasets, the models start performing well using around 1k vulnerable examples. We developed a logistic regression model that can find programs that are difficult for the model to predict correctly. The explanation tools showed that the models used common features to make predictions, ranging from 3.38-6.88 lines in common per top 10 important lines. We report the code patterns that the models frequently highlighted as important features. In the future work, we plan to further investigate these patterns.

VII. Acknowledgements

We thank the anonymous reviewers for their valuable feedback. We thank Hongyang Gao for providing computing resources for our experiments. We thank Qi Li for discussing an experiment metric. This research is partially supported by the U.S. National Science Foundation (NSF) under Award #1816352.

References

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

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation, 2021.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[4] Xinbo Ban, Shigang Liu, Chao Chen, and Caslon Chua. A performance evaluation of deep-learnt features for software vulnerability detection. *Concurrency and Computation: Practice and Experience*, 31(19), October 2019.

[5] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Information and Software Technology*, 136:106576, August 2021.

[6] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. Mvd: Memory-related vulnerability detection based on flow-sensitive graph neural networks. 2022.

[7] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2022.

[8] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Transactions on Software Engineering and Methodology*, 30(3):38:1–38:33, April 2021.

[9] David Coimbra, Sofia Reis, Rui Abreu, Corina Păsăreanu, and Hakan Erdogmus. On using distributed representations of source code for the detection of C security vulnerabilities, 2021.

[10] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations*, 2020.

[11] Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. Towards learning (dis)-similarity of source code from program contrasts, 2021.

[12] Yangruibo Ding, Sahil Suneja, Yunhui Zheng, Jim Laredo, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. Velvet: a novel ensemble learning approach to automatically locate vulnerable statements, 2022.

[13] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A C/C++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 508–512, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020.

[15] Michael Fu and Chakkrit Tantithamthavorn. LineVul: A transformer-based line-level vulnerability prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 608–620, 2022.

[16] Hazim Hanif and Sergio Maffeis. VulBERTa: Simplified source code pre-training for vulnerability detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2022.

[17] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations*, 2020.

[18] David Hin, Andrey Kan, Huaming Chen, and M. Ali Babar. LineVD: Statement-level vulnerability detection using graph neural networks, 2022.

[19] IBM. Project codenet, 2021.

[20] Paul Jaccard. The distribution of the flora in the alpine zone.1. *New Phytologist*, 11(2):37–50, 1912.

[21] Yi Li, Shaohua Wang, and Tien N. Nguyen. 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*, ESEC/FSE 2021, page 292–303, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. VulDeeLocator: A deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2821–2837, jul 2022.

[23] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, jul 2022.

[24] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, 2018.

[25] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software Vulnerability Detection Using Deep Neural Networks: A Survey. *Proceedings of the IEEE*, 108(10):1825–1848, October 2020. Conference Name: Proceedings of the IEEE.

[26] Guanjun Lin, Wei Xiao, Leo Yu Zhang, Shang Gao, Yonghang Tai, and Jun Zhang. Deep neural-based vulnerability discovery demystified: data, model and performance. *Neural Computing and Applications*, 33(20):13287–13300, October 2021.

[27] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

[28] Alejandro Mazuera-Rozo, Anamaria Mojica-Hanke, Mario Linares-Vásquez, and Gabriele Bavota. Shallow or Deep? An Empirical Study on Detecting Vulnerabilities using Deep Learning. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 276–287, May 2021. ISSN: 2643-7171.

[29] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[30] Deep Mind. Competitive programming with alphacode, 2022.

[31] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. ReGVD: Revisiting graph neural

networks for vulnerability detection. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 178–182, 2022.

[32] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. CoTexT: Multi-task learning with code-text transformer, 2021.

[33] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why Should I Trust You?": Explaining the predictions of any classifier. KDD '16, page 1135–1144, New York, NY, USA, 2016. Association for Computing Machinery.

[34] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. *Proceedings - 17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018*, pages 757–762, 2019. Publisher: IEEE.

[35] Zihua Song, Junfeng Wang, Shengli Liu, Zhiyang Fang, Kaiyuan Yang, and Gu Zhaoquan. HGVul: A code vulnerability detection method based on heterogeneous source-level intermediate representation. *Sec. and Commun. Netw.*, 2022, jan 2022.

[36] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986.

[37] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 3319–3328. JMLR.org, 2017.

[38] Gaigai Tang, Lianxiao Meng, Huiqiang Wang, Shuangyin Ren, Qiang Wang, Lin Yang, and Weipeng Cao. A Comparative Study of Neural Network Techniques for Automatic Software Vulnerability Detection. In *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–8, December 2020.

[39] Ian Tenney, James Wexler, Jasmijn Bastings, Tolga Bolukbasi, Andy Coenen, Sebastian Gehrmann, Ellen Jiang, Mahima Pushkarna, Carey Radebaugh, Emily Reif, and Ann Yuan. The language interpretability tool: Extensible, interactive visualizations and analysis for NLP models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 107–118, Online, October 2020. Association for Computational Linguistics.

[40] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. *ACM SIGPLAN Notices*, 44(6):87–97, June 2009.

[41] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation, September 2021. arXiv:2108.04556 [cs].

[42] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. VulCNN: an image-inspired scalable vulnerability detection system. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 2365–2376, New York, NY, USA, May 2022. Association for Computing Machinery.

[43] Jiaxi Xu, Jun Ai, Jingyu Liu, and Tao Shi. ACGDP: An Augmented Code Graph-Based System for Software Defect Prediction. *IEEE Transactions on Reliability*, pages 1–10, 2022. Conference Name: IEEE Transactions on Reliability.

[44] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. GNN explainer: A tool for post-hoc explanation of graph neural networks. *CoRR*, abs/1903.03894, 2019.

[45] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794, Montreal, QC, Canada, May 2019. IEEE.

[46] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. D2a: A dataset built for ai-based vulnerability detection methods using differential analysis. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems*, 32:1–11, 2019.