Deep Learning Based Vulnerability Detection: Are We There Yet?

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray

Abstract—Automated detection of software vulnerabilities is a fundamental problem in software security. Existing program analysis techniques either suffer from high false positives or false negatives. Recent progress in Deep Learning (DL) has resulted in a surge of interest in applying DL for automated vulnerability detection. Several recent studies have demonstrated promising results achieving an accuracy of up to 95 percent at detecting vulnerabilities. In this paper, we ask, *"how well do the state-of-the-art DL-based techniques perform in a real-world vulnerability prediction scenario?"* To our surprise, we find that their performance drops by more than 50 percent. A systematic investigation of what causes such precipitous performance drop reveals that existing DL-based vulnerability prediction approaches suffer from challenges with the training data (e.g., data duplication, unrealistic distribution of vulnerable classes, etc.) and with the model choices (e.g., simple token-based models). As a result, these approaches often do not learn features related to the actual cause of the vulnerabilities. Instead, they learn unrelated artifacts from the dataset (e.g., specific variable/function names, etc.). Leveraging these empirical findings, we demonstrate how a more principled approach to data collection and model design, based on realistic settings of vulnerability prediction, can lead to better solutions. The resulting tools perform significantly better than the studied baseline—up to 33.57 percent boost in precision and 128.38 percent boost in recall compared to the best performing model in the literature. Overall, this paper elucidates existing DL-based vulnerability prediction systems' potential issues and draws a roadmap for future DL-based vulnerability prediction research.

Index Terms—Vulnerability, deep learning based vulnerability detection, real world vulnerabilities, graph neural network based vulnerability detection

1 Introduction

Automated detection of security vulnerabilities is a fundamental problem in systems security. Traditional techniques are known to suffer from high false-positive/false-negative rates [1], [2]. For example, static analysis-based tools typically result in high false positives detecting non-vulnerable (hereafter, neutral)[1] cases as vulnerable, and dynamic analysis suffers from high false negatives. So far these tools remain unreliable, leaving significant overhead for developers [2].

Recent progress in Deep Learning (DL), especially in domains like computer vision and natural language processing, has sparked interest in using DL to detect security vulnerabilities automatically with high accuracy. According to Google scholar, 92 papers appeared in popular security and software engineering venues between 2019 and 2020 that apply learning techniques to detect different types of bugs.[2] In fact, several recent studies have demonstrated very promising results achieving accuracy up to 95 percent [3], [4], [5], [6].

Given such remarkable reported success of DL models at detecting vulnerabilities, it is natural to ask why they are performing so well, what kind of features these models are learning, and whether they are generalizable, i.e., can they be used to reliably detect real-world vulnerabilities?

The generalizability of a DL model is often limited by implicit biases in the dataset, which are often introduced during the dataset generation/curation/labeling process and therefore affect both the testing and training data equally (assuming that they are drawn from the same dataset). These biases tend to allow DL models to achieve high accuracy in the test data by learning highly idiosyncratic features specific to that dataset instead of generalizable features. For example, Yudkowsky *et al.* [7] described an instance where US Army found out that a neural network for detecting camouflaged tanks did not generalize well due to dataset bias even though the model achieved very high accuracy in the testing data. They found that all the photos with the camouflaged tanks in the dataset were shot in cloudy days, and the model simply learned to classify lighter and darker images instead of detecting tanks.

In this paper, we systematically measure the generalizability of four state-of-the-art Deep Learning-based Vulnerability Prediction (hereafter DLVP) techniques [3], [4], [5], [6] that have been reported to detect security vulnerabilities with high accuracy (up to 95 percent) in the existing literature. We primarily focus on the Deep Neural Network (DNN) models that take source code as input [3], [4], [5], [6], [8] and detect vulnerabilities at function granularity.

1. We prefer to refer to non-vulnerable code as "neutral" to indicate that they contain no *known* vulnerabilities or that they do not fall in any known vulnerability category.

2. published in TSE, ICSE, FSE, ASE, S&P Oakland, CCS, USENIX Security, etc.

• *The authors are with the Columbia University, New York, NY 10027 USA. E-mail: {saikatc, rayb}@cs.columbia.edu, i.m.ralk@gmail.com, yangruibo.ding@columbia.edu.*

Manuscript received 3 Sept. 2020; revised 17 May 2021; accepted 20 May 2021. Date of publication 8 June 2021; date of current version 19 Sept. 2022.
(Corresponding author: Rahul Krishna.)
Recommended for acceptance by L. Mariani.
Digital Object Identifier no. 10.1109/TSE.2021.3087402

0098-5589 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.

These models operate on a wide range of datasets that are either generated synthetically or adapted from real-world code.

First, we curate a new vulnerability dataset from two large-scale popular real-world projects (Chromium and Debian) to evaluate the performance of existing techniques in the real-world vulnerability prediction setting. The code samples are annotated as vulnerable/neutral, leveraging their issue tracking systems. Since both the code and annotations come from the real-world, detecting vulnerabilities using such a dataset reflects a realistic vulnerability prediction scenario. We also use FFMPeg+Qemu dataset proposed by Zhou *et al.* [6].

To our surprise, we find that none of the existing models perform well in real-world settings. If we directly use a pre-trained model to detect the real-world vulnerabilities, the performance drops by $\sim$*73%*, on average. Even if we retrain these models with real-world data, their performance drops by $\sim$*54%* from the reported results. For example, VulDeePecker [3] reported a precision of 86.9 percent in their paper. However, when we use VulDeePecker's pre-trained model in real world datasets, its precision reduced to 11.12 percent, and after retraining, the precision becomes 17.68 percent. A thorough investigation revealed the following problems:

- *Inadequate Model.* The most popular models treat code as a sequence of tokens and do not take into account semantic dependencies that play a vital role in vulnerability predictions. Further, when a graph-based model is used, it does not focus on increasing the class-separation between vulnerable and neutral categories. Thus, in realistic scenarios, they suffer from low precision and recall.
- *Learning Irrelevant Features.* Using state-of-the-art explanation techniques [9], we find that the current models are essentially learning up irrelevant features that are not related to vulnerabilities and are likely artifacts of the datasets.
- *Data Duplication.* The training and testing data in most existing approaches contain duplicates (up to *68 percent*); thus, artificially inflating the reported results.
- *Data Imbalance.* Existing approaches do not alleviate the class imbalance problem [10] of real-world vulnerability distribution where neutral code are more frequent than the vulnerable code.

To overcome these problems, we propose a road-map that we hope will help the DL-based vulnerability prediction researchers to avoid such pitfalls in the future. To this end, we demonstrate how a more principled approach to data collection and model design, based on our empirical findings, can lead to better solutions. For data collection, we discuss how to curate real-world vulnerability prediction data incorporating both static and evolutionary (i.e., bug-fix) nature of the vulnerabilities. For model building, we show representation learning [11] can be used on top of traditional DL methods to increase the class separation between vulnerable and neutral samples.

We further empirically establish that the use of semantic information (with graph-based models), data de-duplication, and balancing training data can significantly improve vulnerability prediction. Following these steps, we can boost precision and recall of the best performing model in the literature by up to 33.57 and 128.38 percent respectively.

In short, this paper argues that DL-based vulnerability detection is still very much an open problem and requires a well-thought-out data collection and model design framework guided by real-word vulnerability detection settings. To this end, we make the below contributions:

(1) We systematically study existing approaches in DLVP task and identify several problems with the current dataset and modeling practices.
(2) Leveraging the empirical results, we propose a summary of best practices that can help future DLVP research and experimentally validate these suggestions.
(3) We curated a real-world dataset from developer/user reported vulnerabilities of Chromium and Debian projects (Available at https://bit.ly/3bX30ai).
(4) We also open source all our code and data we used in this study for broader dissemination. Our code and replication data are available in https://git.io/Jf6IA.

2 Background and Challenges

DLVP methods aim to detect unknown vulnerabilities in target software by learning different vulnerability patterns from a training dataset. Most popular DLVP approaches consist of three steps: data collection, model building, and evaluation. First, data is collected for training, and an appropriate model is chosen as per design goal and resource constraints. The training data is preprocessed according to the format preferred by the chosen model. Then the model is trained to minimize a loss function. The trained model is intended to be used in the real world. To assess the effectiveness of the model, performance is evaluated on unseen test examples.

This section describes the theory of DL-based vulnerability prediction approaches (Section 2.1), existing datasets (Section 2.2), existing modeling techniques (Section 2.3), and evaluation procedure (Section 2.4). Therein, we discuss the challenges that potentially limit the applicability of existing DLVP techniques.

2.1 DLVP Theory

DL-based vulnerability predictors learn the vulnerable code patterns from a training data (D_{train}) set where code elements are labeled as vulnerable or neutral. Given a code element (x) and corresponding vulnerable/neutral label (y), the goal of the model is to learn features that maximize the probability $p(y|x)$ with respect to the model parameters (θ). Formally, training a model is learning the optimal parameter settings (θ^*) such that

$$\theta^* = argmax_\theta \prod_{(x,y)\in D_{train}} p(y|x,\theta). \quad (1)$$

First, a code element (x^i) is transformed to a real valued vector ($h^i \in \mathbb{R}^n$), which is a compact representation of x^i. How a model transforms x^i to h^i depends on the specifics of the model. This h^i is transformed to a scalar $\hat{y} \in [0,1]$ which denotes the probability of code element x^i being vulnerable.

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.

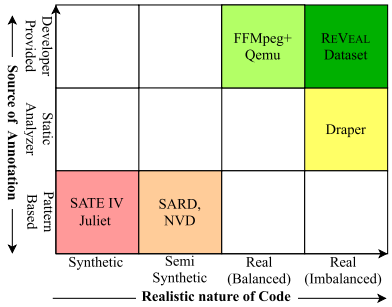


Fig. 1. Different DLVP dataset and their synthetic/realistic nature. From red to *green*, colors symbolize increasing realistic nature of dataset. Red is the most synthetic, *green* is the most realistic.

In general, this transformation and probability calculation is achieved through a feed forward layer and a softmax [12] layer in the model. Typically, for binary classification task like vulnerability prediction, optimal model parameters are learned by minimizing the cross-entropy loss [13]. Cross-entropy loss penalizes the discrepancy in the model's predicted probability and the actual probability (0. for neutral 1. for vulnerable examples) [14].

2.2 Existing Dataset

To train a vulnerability prediction model, we need a set of annotated code that are labeled vulnerable or neutral. The number of vulnerable code should be large enough to allow the model to learn from it. Researchers used a variety of data sources for DLVP (see Fig. 1). Depending on how the code samples are collected and how they are annotated, we classify them as:

- *Synthetic data:* The vulnerable code example and the annotations are artificially created. SATE IV Juliet [15] dataset and SARD [16] fall in this category. Here the examples are synthesized using known vulnerable patterns. These datasets were originally designed for evaluating static and dynamic analysis based vulnerability prediction tools.
- *Semi-synthetic data:* Here either the code or the annotation is derived artificially. For example, Draper dataset, proposed by Russell *et al.* [5], contains functions that are collected from open source repositories but are annotated using static analyzers. Examples of SARD [16] and National Vulnerability Database (NVD [17]) dataset are also taken from production code; however, they are often modified in a way to demonstrate the vulnerability isolating them from their original context. Although these datasets are more complex than synthetic ones, they do not fully capture the complexities of the real-world vulnerabilities due to simplifications and isolations.
- *Real data:* Here both the code and the corresponding vulnerability annotations are derived from real-world sources. For instance, Zhou *et al.* [6] curated *Devign* dataset, which consists of past vulnerabilities and their fixes from four open-source projects, two of which are publicly available.

Problem 1: Data source and their annotation are unrealistic. Fig. 1) compares current vulnerability datasets in terms of the realism of code (x-axis) and the annotation strategy (Y-axis). A model trained on a synthetic dataset, i.e., those comprising of simple and unrealistic code examples, will be limited to detecting only those simple patterns and they seldom occur in real life vulnerabilities.

```
1  void action(char *data) const {
2  // FLAW: Increment of pointer in the loop will cause
3  // freeing of memory not at the start of the buffer.
4     for (; *data != '\0'; data++){
5        if (*data == SEARCH_CHAR){
6           printLine("We have a match!");
7           break;
8        }
9     }
10    free(data);
11 }
```

Fig. 2. Example Vulnerability (CWE761) [18].

As an example, consider a typical buffer overflow example in Fig. 2 used by VulDeePecker and SySeVR. Albeit a good pedagogical example, real world vulnerabilities are not as simple or as isolated as this example. In contrast, Fig. 3 shows another buffer overflow example from Linux kernel. Although the fix here was straightforward, finding the source of the vulnerability requires an in-depth understanding of the semantics of different components of the code such as the variables and the functions. A model is trained to reason about simpler examples as in Fig. 2 will fail to reason about the vulnerability in Fig. 3 code. In addition, any model that is built on data annotated by a static analyzer [5] would inherit all the drawbacks of static analysis such as the high false positive rates [1], [2] and would consequently be severely biased.

Problem 2: The distribution of vulnerable and neutral examples are unrealistic. In the most realistic dataset available to us, FFMPeg+Qemu [6], the ratio of vulnerable and neutral examples is approximately 45-55 percent. However, this does not reflect a real world distribution of vulnerable code. In reality, neutral code far outnumbers vulnerable code examples. Furthermore, this dataset annotates vulnerability by inferring the of commit message of that code goes. It does not discriminate between vulnerable and neutral code, instead it discriminates between the *nature of fix*, i.e., between a vulnerability fixing commit and other commits. A model trained on such a dataset may not perform well in a realistic use case scenario where we need to differentiate

```
1  static void eap_request(
2        eap_state *esp, u_char *inp, int id, int len) {
3    ...
4    if (vallen < 8 || vallen > len) {
5         ...
6         break;
7    }
8  /* FLAW: 'rhostname' array is vulnerable to overflow.*/
9  -   if (vallen >= len + sizeof (rhostname)){
10 +   if (len - vallen >= (int)sizeof (rhostname)){
11       ppp_dbglog(...);
12       MEMCPY(rhostname, inp + vallen,
13            sizeof(rhostname) - 1);
14       rhostname[sizeof(rhostname) - 1] = '\0';
15       ...
16    }
17    ...
18 }
```

Fig. 3. CVE-2020-8597 - A partial patch (original patch [19]) for an instance of buffer overflow vulnerability in Linux point to point protocol daemon (pppd) due to a logic flaw in the packet processor [20], [21].

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.

the vulnerable function from all other neutral functions in it's proximity.

In summary, we seek vulnerability detection tools that can used in real world vulnerability detection and those that are trained on realistic use cases. To that end, we seek to build *evaluation dataset* that closely resemble the complexity of real code and also the skewed distribution of vulnerabilities in projects. This is in contrast with exiting techniques that often simplify real code to isolate vulnerabilities and test static analyzers [16], [22].

2.3 Existing Modeling Approaches

Model selection depends primarily on the information that one wants to incorporate. The popular choices for DLVP are token-based or graph-based models, and the input data (code) is preprocessed accordingly [3], [5], [6].

- *Token-based models:* In the token-based models, code is considered as a sequence of tokens. Existing token-based models used different Neural Network architectures. For instance, Li *et al.* [3] proposed a Bidirectional Long Short Term Memory (BSLTM) based model, Russell *et al.* [5] proposed a Convolutional Neural Network (CNN) and Radom Forest-based model and compared against Recurrent Neural Network (RNN) and CNN based baseline models for vulnerability prediction. For these relatively simple token-based models, token sequence length is an important factor to impact performance as it is difficult for the models to reason about long sequences. To address this problem, VulDeePecker [3] and SySeVR [4] extract code slices. The motivation behind slicing is that not every line in the code is equally important for vulnerability prediction. Therefore, instead of considering the whole code, only slices extracted from "interesting points" in code (e.g., API calls, array indexing, pointer usage, etc.) are considered for vulnerability prediction and rest are omitted.
- *Graph-based models:* These models consider code as graphs and incorporate different syntactic and semantic dependencies. Different type of syntactic graph (Abstract Syntax Tree) and semantic graph (Control Flow graph, Data Flow graph, Program Dependency graph, Def-Use chain graph etc.) can be used for vulnerability prediction. For example, Devign [6] leverage code property graph (CPG) proposed by Yamaguchi *et al.* [8] to build their graph based vulnerability prediction model. CPG is constructed by augmenting different dependency edges (i.e., control flow, data flow, def-use, etc.) to the code's Abstract Syntax Tree (AST) (see Section 4).

Both graph and token-based models have to deal with *vocabulary explosion* problem—the number of possible identifiers (variable, function name, constants) in code can be virtually infinite, and the models have to reason about such identifiers. A common way to address this issue is to replace the tokens with abstract names [3], [4]. For instance, VulDeePecker [3] replaces most of the variable and function names with symbolic names (`VAR1`, `FUNC1`, `VAR2` etc.).

Expected input for all the models are real valued vectors commonly known as embeddings. There are several ways

```
1  void action(char *data) const {
2      for (; *data != '\0'; data++){
3          foo(data);
4          bar(data);
5          if (*data == SEARCH_CHAR){
6              printLine("We have a match!");
7              break;
8          }
9      }
10     free(data);
11 }
```

Fig. 4. Example of CWE-761 [25]. A buffer is freed not at the start of the buffer but somewhere in the middle of the buffer. This can cause the application to crash, or in some cases, modify critical program variables or execute code. This vulnerability can be detected with data dependency.

to embed tokens to vectors. One such way is to use an embedding layer [23] that is jointly trained with the vulnerability prediction task [5]. Another option is to use external word embedding tool(e.g., *Word2Vec* [24]) to create vector representation of every token. VulDeePecker [3] and SySeVR [4] uses *Word2Vec* to transform their symbolic tokens into vectors. Devign [6], in contrast, uses *Word2Vec* to transform the concrete code tokens to real vectors.

Once a model is chosen and appropriate preprocessing is done on the training dataset, the model is ready to be trained by minimizing a loss function. Most of the existing approaches optimize the model by minimizing some variation of cross-entropy loss. For instance, Russell *et al.* [5] optimized their model using cross-entropy loss, Zhou *et al.* [6] used regularized cross entropy loss.

Problem 3: Token-based models lack syntactic representativeness. Token based models assume that tokens are linearly dependent on each other, and thus, only lexical dependencies between the tokens are present, while the semantic dependencies are lost, which often play important roles in vulnerability prediction [26]. To incorporate some semantic information, VulDeePecker [3] and SySeVR [4] extracted program slices of a potentially interesting point. For example, consider the code in Fig. 4. A slice *w.r.t.* `free` function call at line 10 gives us all the lines except lines 6 and 7. The token sequence of the slice are: `void action ( char * data ) const { for ( data ; * data != '\0' ;` **`data ++`**`) { foo ( data ) ; bar ( data ) ; if ( * data == SEARCH_CHAR ) {` **`free ( data ) ;`**. In this examples, while the two main components for this code being vulnerable, i.e. **data ++** (line 2) and **free (data)** (line 10) are present in the token sequence, they are far apart from each other without explicitly maintaining any dependencies.

In contrast, as a graph based model can consider the data dependency edges (red edge), we see that there is a direct edge between those lines making those lines closer to each other making it easier for the model to reason about that connection. Note that this is a simple CWE example (CWE 761), which requires only the data dependency graph to reason about. Real-world vulnerabilities are much more complex and require reasoning about control flow, data flow, dominance relationship, and other kinds of dependencies between code elements [8]. However, graph-based models, in general, are much more expensive than their token-based counterparts and do not perform well in a resource-constrained environment.

Problem 4: Current models are "brittle". Another problem with the existing approaches is that although the trained

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.

models learn to discriminate vulnerable and neutral code samples, the training paradigm does not explicitly focus on increasing the separation between the vulnerable and neutral examples. Thus, with slight variations the classifications become brittle.

Lastly, models suffer from data imbalance [27] between vulnerable and benign code as the proportion of vulnerable examples in comparison to the neutral ones in real world dataset are extremely low [5]. When a model is trained on such imbalanced dataset, models tend to be biased towards the neutral examples.

2.4 Existing Evaluation Approaches

To understand the applicability of a trained model for detecting vulnerability in the real-world, it must first be evaluated. In most cases, a trained model is evaluated on held out test set. Test examples go through the same pre-processing technique as the training and then the model predicts the vulnerability of those pre-processed test examples. This evaluation approach gives an estimate of how the model may perform when used to detect vulnerabilities in the real-world.

Problem 5: The scope of evaluation in current approaches is limited. All the existing approaches report their performances using their own evaluation dataset. Such an evaluation strategy does not give a comprehensive overview of the applicability of the models in other real-world examples. All we can learn from such intra-dataset evaluation is how well their approach fits their own dataset. Although studies report some limited case studies on such on finding vulnerabilities in real-world projects, these case studies do not shed light on the false positives and false negatives [1]. The number of false positives and false negatives are directly correlated to the developer effort in vulnerability prediction [28] and too much of any would hold the developer from using the model [29].

3 ReVeal Data Collection

To address the limitations with the existing data sets (highlighted by *Problems 1* and *2*), we curate a more robust and comprehensive real world dataset, ReVeal, by tracking the past vulnerabilities from two open-source projects: Linux Debian Kernel and Chromium (open source project of Chrome). We select these projects because: (i) these are two popular and well-maintained public projects with large evolutionary history, (ii) the two projects represent two important program domains (OS and browsers) that exhibit diverse security issues, and (iii) both the projects have plenty of publicly available vulnerability reports.

To curate our data, we first collect *already fixed* issues with publicly available patches. For Chromium, we scraped its bug repository Bugzilla.[3] For Linux Debian Kernel, we collected the issues from Debian security tracker.[4] We then identify vulnerability related issues, i.e., we choose those patches that are labeled with "security". This identification mechanism is inspired by the security issue identification techniques proposed in existing literature [30]. Zhou *et al.* [31]'s proposed approach filter out commits that do not have security related keywords.

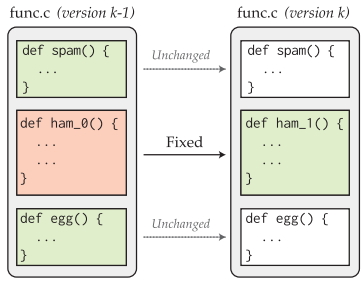


Fig. 5. Collecting real world data for ReVeal. Green samples are labeled as *neutral*, while red sample is marked as *vulnerable*.

For each patch, we extracted the corresponding vulnerable and fixed versions (i.e., old and new version) of `C/C++` source and header files that are changed in the patch. We annotate the previous versions of all changed functions (i.e., the versions prior to the patch) as 'vulnerable' and the fixed version of all the changed functions (i.e., the version after patch) as 'clean'. Additionally, other functions that were not involved in the patch (i.e., those that remained unchanged) are all annotated as 'clean'. Annotating code in this way simulates real-world vulnerability prediction scenario, where a DL model would learn to inspect the vulnerable function in the context of all the other functions in its scope. Further, retaining the fixed variant of the vulnerable function helps the DL model learn the nature of vulnerability-fix patches.

A contrived example of our data collection strategy is illustrated in Fig. 5. Here, we have two versions of a file `file.c`. The previous version of the file (version $k-1$) has a vulnerability which is fixed in the subsequent version (version k) by patching the function `ham_0()` to `ham_1()`. In our dataset, `ham_0()` would be included and labeled 'vulnerable' and `ham_1()` would be included and labeled 'clean'. The other two functions (`spam()` and `egg`) remained unchanged in the patch. Our dataset would include a copy of these two functions and label them as 'clean'. Note, when multiple functions are involved in a vulnerability-fix, we annotate the previous (or pre-commit) version of each changed function as 'vulnerable' and the new (or post-commit) version as 'clean'. Fig. 6 shows a cumulative percentage of vulnerability-fix commit *w.r.t.* the number of vulnerable functions. We observed that, in most cases, the patches change very small number of functions. In 80 percent of the cases, changes spanned 4 or fewer

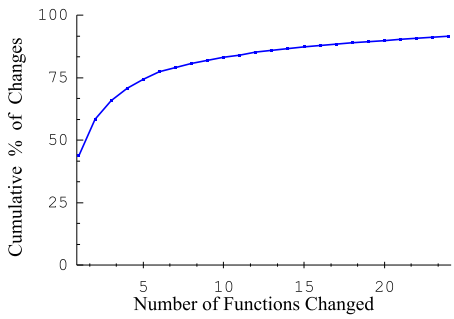


Fig. 6. Number of Functions Changes *w.r.t.* Cumulative percentage of commits.

3. https://bugs.chromium.org/p/chromium/issues/list
4. https://security-tracker.debian.org/tracker/

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Summary of DLVP Datasets and Approaches

Dataset	Used By	# Programs	% Vul*	Granularity	Model Type	Model	Description
SATE IV Juliet [15]	Russell *et al.* [5]	11,896	45.00	Function	Token	CNN+RF	Synthetic code for testing static analyzers.
SARD [16]	VulDeePecker [3]	9,851	31	Slice	Token	BLSTM	Synthetic, academic, and production security flaws or vulnerabilities.
	SySeVR [4]	14,000	13.41	Slice	Token	BGRU	
NVD [17]	VulDeePecker‡	840	31	Slice	Token	BLSTM	Collection of known vulnerabilities from real world projects.
	SySeVR‡	1,592	13.41	Slice	Token	BGRU	
Draper [5]	Russell *et al.* [5]	1,274,366	6.46	Function	Token	CNN+RF	Contains code from public repositories in Github and Debian source repositories.
FFMPeg+Qemu [6]	Devign [6]	22,361	45.02	Function	Graph	GGNN	FFMPeg is a multimedia library; Qemu is hardware virtualization emulator.
ReVeal dataset	This paper	18,169	9.16	Function	Graph	GGNN + MLP + Triplet Loss	Contains code from Chromium and Debian source code repository

**Percentage of vulnerable samples in the dataset.*
‡VulDeePecker and SySeVR uses combination of SARD and NVD datasets to train and evaluate their model.

functions. We would like to acknowledge here that, we considered every commit independent of each other. If a function is modified in two different commits, we annotate *pre-commit* version for both as vulnerables and *post-commit* as clean. We conjecture that it is a realistic evaluation scenario, since while evaluating vulnerability of a function, we would not know what would happen to the function in future.

Table 1 summarizes the details of all the datasets. The details of our ReVeal dataset is highlighted in gray. Additional details of the summary statistics of the ReVeal dataset are tabulated in Table 2.

4 ReVeal: A Roadmap for Vulnerability Prediction

In this section, we present a brief overview of the ReVeal pipeline that aims to lay a *roadmap* for accurately detecting vulnerabilities in real-world projects. Fig. 7 illustrates the ReVeal pipeline. It operates in two phases namely, feature extraction (Phase-I) and training (Phase-II). In the first phase we translate real-world code into a graph-embedding (Section 4.1). In the second phase, we train a representation learner on the extracted features to learn a representation that most ideally demarcates the vulnerable examples from neutral examples (Section 4.2).

4.1 Feature Extraction (Phase-I)

The goal of this phase is to convert code into a compact and a uniform length feature vector while maintaining the semantic and syntactic information. In order to address *Problem 3* (discussed in Section 2.3), our proposed road map extracts a feature vector using a graphical representation of code. Note that, the feature extraction scheme presented below represents the most commonly used series of steps for extracting features from a graph representation [6]. ReVeal uses this scheme to extract the graph embedding of each function in code (graph based feature vector that represent the entirety of a function in a code).

To extract the syntax and semantics in the code, we generate a code property graph (hereafter, CPG) [8]. The CPG is a particularly useful representation of the original code since it offers a combined and a succinct representation of the code consisting of elements from the control-flow and data-flow graph in addition to the AST and program dependency graph (or PDG). Each of the above elements offer additional context about the overall semantic structure of the code [8].

Formally, a CPG is denoted as $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, where V represent the vertices (or nodes) in the graph and E represents the edges. Each vertex V in the CPG is comprised of the vertex type (e.g., `ArithmeticExpression`, `CallStatement` etc.) and a fragment of the original code. To encode the type information, we use a one-hot encoding vector denoted by T_v. To encode the code fragment in the vertex, we use a word2vec embedding denoted by C_v. Next, to create the vertex embedding, we concatenate T_v and C_v into a joint vector notation for each vertex.

The current vertex embedding is not adequate since it considers each vertex in isolation. It therefore lacks information about its adjacent vertices and, as a result, the overall graph structure. This may be addressed by ensuring that each vertex embedding reflects both its information and those of its neighbors. We use gated graph neural networks (hereafter GGNN) [32] for this purpose.

Feature vectors for all the nodes in the graph (X) along with the edges (E) are the input to the GGNN [31], [32]. For every vertex in the CPG, GGNN assigns a gated recurring unit (GRU) that updates the current vertex embedding by assimilating the embedding of all its neighbors. Formally

TABLE 2
Summary Statistics of the ReVeal Dataset

Summary Statistic	Count				
	Min.	25 Q.	Median	75 Q.	Max.
# of lines of code	2	7	15	33	915
# of tokens	7	56	121	274	8830
# of nodes	2	9	18	38	499
# of edges	5	41	87	198	8257
Total Number of Examples					18169
Number of Vulnerable Examples					1658

'25 Q.' and '75 Q.' represents 25th quantile and 75th quantile, respectively. All the statistics are on function granularity.

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.

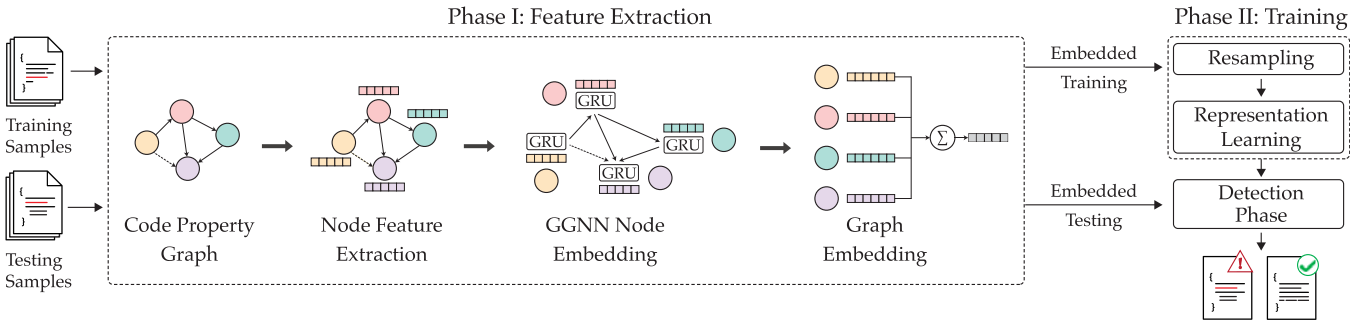


Fig. 7. Overview of the REVEAL vulnerability prediction framework.

$$x'_v = GRU\left(x_v, \sum_{(u,v)\in \mathrm{E}} g(x_u)\right).$$

Where, $GRU(\cdot)$ is a Gated Recurrent Function, x_v is the embedding of the current vertex v, and $g(\cdot)$ is a transformation function that assimilates the embeddings of all of vertex v's neighbors [32], [33], [34]. x'_v is the GGNN-transformed representation of the vertex v's original embedding x_v. x'_v now incorporates v's original embedding x_v as well as the embedding of its neighbors.

The final step in preprocessing is to aggregate all the vertex embedding x'_v to create a single vector representing the whole CPG denoted by x_g, i.e.,

$$x_g = \sum_{v\in \mathrm{V}} x'_v.$$

Note that REVEAL uses a simple element-wise summation as the aggregation function, but in practice it is a configurable parameter in the pipeline. The result of the pipeline presented so far is an $m-$dimensional feature vector representation of the original source code. To pre-train the GGNN, we augment a classification layer on top of the GGNN feature extraction. This training mechanism is similar to Devign [6]. Such pre-training deconstructs the task of "learning code representation", and "learning vulnerability", and is also used by Russell *et al.* [5]. While, we pre-train GGNN in a supervised fashion, unsupervised program representation learning [35] can also be done to learn better program presentation. However, such learning is beyond the scope of this research and we leave that for future research.

4.2 Training (Phase-II)

In real-world data, the number of neutral samples (i.e., negative examples) far outnumbers the vulnerable examples (i.e., positive examples) as shown in Table 1. If left unaddressed, this introduces an undesirable bias in the model limiting its predictive performance. Further, extracted feature vectors of the vulnerable and neutral examples exhibit a significant overlap in the feature space. This makes it difficult to demarcate the vulnerable examples from the neutral ones. Training a DL model without accounting for the overlap makes it susceptible to poor predictive performance.

To mitigate the above problems, we propose a two step approach. First, we use re-sampling to balance the ratio of vulnerable and neutral examples in the training data. Next, we train a representation learning model on the re-balanced data to learn a representation that can most optimally distinguish vulnerable and neutral examples.

4.2.1 Reducing Class Imbalance

In order to handle imbalance in the number of vulnerable and neutral classes, we use the "synthetic minority oversampling technique" (for short, SMOTE) [36]. It operates by changing the frequency of the different classes in the data. Specifically, SMOTE sub-samples the majority class (i.e., randomly deleting some examples) while super-sampling the minority class (by creating synthetic examples) until all classes have the same frequency. In the case of vulnerability prediction, the minority class is usually the vulnerable examples. SMOTE has shown to be effective in a number of domains with imbalanced datasets [37], [38].

During super-sampling, SMOTE picks a vulnerable example and finds k nearest vulnerable neighbors. It then builds a synthetic member of the minority class by interpolating between itself and one of its random nearest neighbors. During under-sampling, SMOTE randomly removes neutral examples from the training set. This process is repeated until a balance is reached between the vulnerable and neutral examples. Note that, while we use off-the-shelf SMOTE for re-balancing training data, other data re-balancing technique (e.g., MWMOTE [39], ProWSyn [40]). Nevertheless, SMOTE as a re-balancing module in REVEAL's pipeline is configurable and can easily be replaced by other re-balancing techniques. Comparison between different re-balancing techniques themselves is beyond the scope of this research.

4.2.2 Representation Learning Model

The graph embedding of the vulnerable and neutral code samples at the end of Phase-I tend to exhibit a high degree of overlap in feature space. This makes the models "brittle" as highlighted previously by *Problem 4* in Section 2. This effect is illustrated by the t-SNE plot [41] of the feature space in Figs. 10a, 10b, 10c, and 10d. In these examples, there are no clear distinctions between the vulnerable (denoted by $+$) and the neutral samples (denoted by $\circ$). This lack of separation makes it particularly difficult to train an ML model to learn the distinction between the vulnerable and the neutral samples.

To improve the predictive performance, we seek a model that can project the features from the original non-separable space into a latent space which offers a better separability

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.

between vulnerable and neutral samples. For this, we use a multi-layer perceptron (MLP) [13], designed to transform input feature vector (x_g) to a latent representation denoted by $h(x_g)$. The MLP consists of three groups of layers namely, the input layer (x_g), a set of intermediate layers which are parameterized by θ (denoted by $f(\cdot, \theta)$, and a final output layer denoted by $\hat{y}$.

The proposed representation learner works by taking as input the original graph embedding x_g and passing it through the intermediate layers $f(\cdot, \theta)$. The intermediate layer project the original graph embedding x_g onto a latent space $h(x_g)$. Finally, the output layer uses the features in the latent space to predict for vulnerabilities as, $\hat{y} = \sigma(\mathrm{W} * h(x_g) + b)$. Where σ represents the softmax function, h_g is the latent representation, W and b represent the model weights and bias respectively.

To maximize the separation between the vulnerable and the neutral examples in the latent space, we adopt the triplet loss [42] as our loss function. Triplet loss has been widely used in machine learning, specifically in representation learning, to create a maximal separation between classes [43], [44]. The triplet loss is comprised of three individual loss functions: (a) cross entropy loss ($\mathcal{L}_{CE}$); (b) projection loss ($\mathcal{L}_p$); and (c) regularization loss ($\mathcal{L}_{reg}$). It is given by:

$$\mathcal{L}_{trp} = \mathcal{L}_{CE} + \alpha * \mathcal{L}_p + \beta * \mathcal{L}_{reg}, \tag{2}$$

α and β are two hyperparameters indicating the contribution of projection loss and regularization loss respectively. The first component of the triplet loss is to measure the cross-entropy loss to penalize miss-classifications. Cross-entropy loss increases as the predicted probability diverges from the actual label. It is given by

$$\mathcal{L}_{CE} = -\sum \hat{y} \cdot log(y) + (1 - \hat{y}) \cdot log(1 - y). \tag{3}$$

Here, y is the true label and $\hat{y}$ represents the predicted label. The second component of the triplet loss is used the quantify how well the latent representation can separate the vulnerable and neutral examples. A latent representation is considered useful if all the vulnerable examples in the latent space are close to each other while simultaneous being farther away from all the neutral examples, i.e., examples from same class are very close (i.e., similar) to each other and examples from different class are far away from each other. Accordingly, we define a loss function $\mathcal{L}_p$ which is defined by

$$\mathcal{L}_p = \big|\mathbb{D}(h(x_g), h(x_{same})) - \mathbb{D}(h(x_g), h(x_{diff})) + \gamma\big|. \tag{4}$$

Here, $h(x_{same})$ is the latent representation of an example that belongs to the same class as x_g and $h(x_{diff})$ is the latent representation of an example that belongs to a different class as that of x_g. Further, γ is a hyperparameter used to define a minimum separation boundary. Lastly, $\mathbb{D}(\cdot)$ represents the cosine distance between two vectors and is given by

$$\mathbb{D}(v_1, v_2) = 1 - \left|\frac{v_1.v_2}{||v_1|| * ||v_2||}\right|. \tag{5}$$

If the distance between two examples that belong to the same class is large (i.e., $\mathbb{D}(h(x_g), h(x_{same}))$ is large) or if the distance between two examples that belong to different classes is small (i.e., $\mathbb{D}(h(x_g), h(x_{diff}))$ is small), $\mathcal{L}_p$ would be large to indicate a sub-optimal representation.

The final component of the triplet loss is the regularization loss ($\mathcal{L}_{reg}$) that is used to limit the magnitude of latent representation ($h(x_g)$). It has been observed that, over several iterations, the latent representation $h(x_g)$ of the input x_g tend to increase in magnitude arbitrarily [45]. Such arbitrary increase in $h(x_g)$ prevents the model from converging [46]. Therefore, we use a regularization loss ($\mathcal{L}_{reg}$) to penalize latent representations ($h(x_g)$) that are larger in magnitude. The regularization loss is given by:

$$\mathcal{L}_{reg} = ||h(x_g)|| + ||h(x_{same})|| + ||h(x_{diff})||. \tag{6}$$

With the triplet loss function, ReVeal trains the model to optimize for it parameters (i.e., θ, W, b) by minimizing Equation (2). The effect of using representation learning can be observed by the better separability of the vulnerable and neutral examples in Fig. 10b.

5 Experimental Setup

5.1 Implementation Details

We use Pytorch 1.4.0 with Cuda version 10.1 to implement our method. For GGNN, we use tensorflow 1.15. We ran our experiments on single Nvidia Geforce 1080Ti GPU, Intel (R) Xeon(R) 2.60GHz 16 CPU with 252 GB ram. Neither Devign's implementation, nor their hyperparameters are not publicly availavle. We followed their paper and re-implemented to our best ability. For the GGNN, maximum iteration number is set to be 500. For the representation learner maximum iteration is 100. We stop the training procedure if F1-score on validation set does not increase in for 50 consecutive training iteration for GGNN and 5 for Representation Learning.

5.2 Study Subject

Table 1 summarizes all the vulnerability prediction approaches and datasets studied in this paper. We evaluate the existing methods (i.e., VulDeePecker [3], SySeVR [4], Russell *et al.* [5], and Devign [6]) and ReVeal's performance on two real world datasets (i.e., ReVeal dataset, and FFMPeg+Qemu). FFMPeg+Qemu was shared by Zhou *et al.* [6] who also proposed the Devign model in the same work. Their implementation of Devign was not publicly available. We re-implement their method to report our results. We ensure that our results closely match their reported results in identical settings.

5.3 Evaluation

To understand a model's performance, researchers and model developers need to understand the performance of a model against a known set of examples. However, as noted by *Problem 5* in Section 2, current evaluation strategies are limited both in scope and in their choice of evaluation metrics. There are two important aspects to consider during evaluation, (a) the evaluation metric, and (b) the evaluation procedure.

Problem Formulation and Evaluation Metric. Most of the approaches formulate the problem as a classification

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.

problem, where given a code example, the model will provide a binary prediction indicating whether the code is vulnerable or not. This prediction formulation relies on the fact that there are sufficient number of examples (both vulnerable and neutral) to train on. In this study, we are focusing on the similar formulation. While both VulDeePecker and SySeVR formulate the problem as classification of code slices, we followed the problem formulation used by Russell *et al.* [5], and Devign [31], where we classify the function. We note that slices are paths in the control/data flow/dependency graphs, and a slice lacks the rich connectivity of nodes that is present in the whole graph. Thus we chose to classify the whole graph in contrast to the slices.

We study approaches based on four popular evaluation metrics for classification task [47] – Accuracy, Precision, Recall, and F1-score. Precision, also known as Positive Predictive rate, is calculated as *true positive / (true positive + false positive)*, indicates correctness of predicted vulnerable samples. Recall, on the other hand, indicates the effectiveness of vulnerability prediction and is calculated as *true positive / (true positive + false negative)*. F1-score is defined as the geometric mean of precision and recall and indicates balance between those.

Evaluation Procedure. Since DL models highly depend on the randomness [48], to remove any bias created due to the randomness, we run 30 trials of the same experiment. At every run, we randomly split the dataset into disjoint train, validation, and test sets with 70, 10, and 20 percent of the dataset respectively. We report the median performance and the inter-quartile range (IQR) of the performance. When comparing the results to baselines, we use statistical significance test [49] and effect size test [50]. Significance test tells us whether two series of samples differ merely by random noises. Effect sizes tells us whether two series of samples differ by more than just a trivial amount. To assert statistically sound comparisons, following previous approaches [51], [52], we use a non-parametric bootstrap hypothesis test [53] in conjunction with the A12 effect size test [54]. We distinguish results from different experiments if both significance test and effect size test agreed that the division was statistically significant (99 percent confidence) and is not result of a "small" effect (A12 $\geq$ 60%) (similar to Agrawal *et al.* [51]).

6 Empirical Results

6.1 Effectiveness of Existing Vulnerability Prediction Approaches (RQ1)

Motivation. The goal of any DLVP approaches is to be able to predict vulnerabilities in the real-world. The datasets that the existing models are trained on contain simplistic examples that are representative of real-world vulnerabilities. Therefore, we ought to, in theory, be able to use these models to detect vulnerabilities in the real-world.

Approach. There are two possible scenarios under which these models may be used:

- *Scenario-A (pre-trained models)*: We may reuse the existing pre-trained models as it is to predict real-world vulnerabilities. To determine how they perform in such a setting, we first train the baseline models with their respective datasets as per Table 1. Next, we use those pre-trained models to detect vulnerabilities in the real-world (i.e., on FFMPeg +Qemu, and ReVeal dataset).
- *Scenario-B (re-trained models)*: We may rebuild the existing models first by training them on the real-world datasets, and then use those models to detect the vulnerabilities. To assess the performance of baseline approaches in this setting, we first use one portion of the FFMPeg+Qemu and ReVeal dataset to train each model. Then, we use those models to predict for vulnerabilities in the remainder of the FFMPeg+Qemu and ReVeal. We repeat the process 30 times, each time training and testing on different portions of the dataset.

Observations. Table 3b tabulates the performance of existing pre-trained models on predicting vulnerabilities in real-world data (i.e., Scenario-A). We observe a precipitous drop in performance when pre-trained models are used for real-world vulnerability prediction.

For example, In ReVeal dataset, VulDeePecker achieves an F1-score of *only* 12.18% and in FFMPeg+Qemu, VulDeePecker achieves an F1-score of 14.27%, while in the baseline case (see Table 3a), the F1-score of VulDeePecker was as high as 85.4%. Even the sophisticated graph-based Devign model produced an F1-score of only ~17% and precision as low as ~10% on ReVeal dataset. Similar performance drops are observed for all the other baselines. On average, we observe a 73 percent drop of F1-score across all the models in this setting.

For scenario-B, Table 3c tabulates our findings for re-trained models. Here, we also observe a significant performance drop from the baseline results. In ReVeal dataset, both Russell *et al.* and VulDeePecker achieve an F1-score of roughly 15% (in contrast to their baseline performances of 85%). SySeVR achieved an F1-score of 30% on ReVeal dataset. We observed similar trends in other settings, with an average F1 score drop of 54 percent.

Result. Existing approaches fail to generalize to real-world vulnerability prediction. If we directly use a pre-trained model to detect the real-world vulnerabilities, the f1-score drops by ~73%, on average. Even if we retrain these models with real-world data, their performance drops by ~54% from the reported results.

6.2 Key Limitations of Existing DLVP Approaches (RQ2)

Motivation. In this RQ, we investigate the reasons behind their failure. We find that the baseline methods suffer from a number of problems, as listed below:

6.2.1 Data Duplication

Preprocessing techniques such as slicing used by VulDeePecker and SySeVR and tokenization used by Russell *et al.* introduce a large number of duplicates in both the training and testing data. There are several ways duplication can be introduced by these preprocessing techniques – e.g., same slice can be extracted from different entry points, different code can have same tokens due to the abstract tokenization, etc.

Let us consider the example in Fig. 8. The statement `read()` (line 5) is the API call of interest for SySeVR. The extracted slice will consist of **red** colored lines. Now, when

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.

TABLE 3
Performance of Existing Approaches in Predicting Real World Vulnerability

(a) Baseline scores reported by the respective papers. We report single values since authors do not report Median *(IQR)*.

Dataset	Technique	Training	Acc	Prec	Recall	F1
Baseline	VulDeePecker	NVD/SARD	·	86.90	·	85.40
	SySeVR	NVD/SARD	95.90	82.50	·	85.20
	Russell *et al.*	Juliet	·	·	·	84.00
		Draper	·	·	·	56.6
	Devign	FFMPeg+Qemu	72.26	·	·	73.26

· = Not Reported.

(b) Scenario-A: Using Existing Pre-trained Models

Dataset	Technique	Training	Acc	Prec	Recall	F1
ReVeal dataset	VulDeePecker	NVD/SARD	79.05 *(0.25)*	11.12 *(0.48)*	13.64 *(0.50)*	12.18 *(0.47)*
	SySeVR	NVD/SARD	79.48 *(0.24)*	9.38 *(0.30)*	15.89 *(0.63)*	10.37 *(0.36)*
	Russell *et al.*	Juliet	38.11 *(0.11)*	41.36 *(0.38)*	6.51 *(0.07)*	11.24 *(0.12)*
		Draper	70.08 *(0.14)*	49.05 *(0.35)*	15.61 *(0.12)*	23.66 *(0.24)*
	Devign	FFMPeg+Qemu	66.24 *(0.14)*	10.74 *(0.11)*	37.04 *(0.54)*	16.68 *(0.17)*
FFMpeg + Qemu	VulDeePecker	NVD/SARD	52.27 *(0.23)*	8.51 *(0.22)*	44.78 *(0.66)*	14.27 *(0.33)*
	SySeVR	NVD/SARD	52.52 *(0.18)*	10.62 *(0.22)*	46.69 *(0.20)*	16.77 *(0.31)*
	Russell *et al.*	Juliet	49.84 *(0.10)*	33.17 *(0.13)*	45.53 *(0.14)*	37.65 *(0.12)*
		Draper	53.96 *(0.14)*	44.00 *(0.17)*	49.53 *(0.20)*	46.60 *(0.15)*

(c) Scenario-B: Using Retrained Models with Real-world Data.

Dataset	Input	Approach	Acc	Prec	Recall	F1
ReVeal dataset	Token	Russell *et al.*	90.98 *(0.75)*	24.63 *(5.35)*	10.91 *(2.47)*	15.24 *(2.74)*
	Slice + Token	VulDeePecker	89.05 *(0.80)*	17.68 *(7.51)*	13.87 *(8.53)*	15.7 *(6.41)*
		SySeVR	84.22 *(2.48)*	24.46 *(4.85)*	40.11 *(4.71)*	30.25 *(2.35)*
	Graph	Devign	88.41 *(0.66)*	34.61 *(3.24)*	26.67 *(6.01)*	29.87 *(4.34)*
FFMpeg + Qemu	Token	Russell *et al.*	58.13 *(0.88)*	54.04 *(2.09)*	39.50 *(2.17)*	45.62 *(1.33)*
	Slice + Token	VulDeePecker	53.58 *(0.61)*	47.36 *(1.80)*	28.70 *(12.08)*	35.20 *(8.82)*
		SySeVR	52.52 *(0.81)*	48.34 *(1.51)*	65.96 *(7.12)*	56.03 *(3.20)*
	Graph	Devign†	58.57 *(1.03)*	53.60 *(3.21)*	62.73 *(2.99)*	57.18 *(2.58)*

All the numbers are reported as Median (IQR) *format.*
†*We made several unsuccessful attempts to contact the authors for Devign's implementation. Despite our best effort, Devign's reported result is not reproducible. We make our implementation of Devign public at* https://github.com/saikat107/Devign for further use.

SySeVR extracts a slice from the code in Fig. 8 based on the arithmetic expression (say `product = product * i;` in line 8), that slice will essentially be the same as the slice extracted from the API call on line 5. This will create a duplicate. Additionally, their tokenization method (replacing concrete identifiers with abstract ones like `VAR_1`, `FUNC_1`, etc.) creates more duplicates between train and test splits.

```
1  void foo(){
2     int i = 0;
3     int sum = 0;
4     int product = 1;
5     int w = read();
6     for(i = 1; i < N; ++i) {
7        sum = sum + i + w;
8        product = product * i;
9     }
10    print(sum);
11 }
```

Fig. 8. Example code demonstrating duplication by slice.

TABLE 4
Percentage of Duplicate Samples in Datasets

Dataset	Pre-processing Technique	% of duplicates
Juliet	Russell *et al.*	68.63
NVD + SARD	VulDeePecker	67.33
	SySeVR	61.99
Draper	Russell *et al.*	6.07 / 2.99
ReVeal dataset	None	0.6
	VulDeePecker	25.85
	SySeVR	25.56
	Russell *et al.*	8.93
FFMPeg+Qemu	None	0.2
	VulDeePecker	19.58
	SySeVR	22.10
	Russell *et al.*	20.54

Approach. We apply each preprocessing technique to its respective dataset (see Section 2) and also to the real-world datasets.

Observations. Table 4 tabulates the number of duplicates introduced by some of the vulnerability prediction approaches. We observe that the preprocessing technique of SySeVR and VulDeePecker (i.e., slicing followed by tokenization) introduces a significant amount of (>60%) duplicate samples. Further, semi-synthetic datasets like NVD, SARD, and Juliet (comprised of much simpler code snippets) result in a large number of duplicates. In contrast, real-world datasets are much more complex and therefore have far fewer duplicates. In our case, the two real-world data contain little to no duplicates prior to preprocessing (ReVeal dataset had only 0.6 percent, and FFMPeg+Qemu had 0.2 percent). After preprocessing, although some duplicates are introduced (e.g., SySeVR's preprocessing technique introduces 25.56 percent duplicates in ReVeal dataset and 22.10 percent duplicates in FFMPeg+Qemu), they are much lesser than baseline datasets. While duplicates created by slicing and pre-processing techniques do favor vulnerability prediction in general [4], [55], it seriously undermines the capability of a DL model to extract patterns. In fact, prevalence of such duplicates in training set might lead a DL model to learn irrelevant features. Common examples between train and test sets hampers fair comparison of different DL models for vulnerability prediction task.

Ideally, a DL based model should be trained and tested on a dataset where 100 percent examples are unique. Inter-

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.

set duplicates may cause the model to memorize examples from training set, making the performance measure very unreliable. Intra-set duplicates (sepcially duplicates inside test set) tend to artificially inflate the overall performance of a method [56], as evidenced by the discrepancy of the baseline results and results of the pre-trained models in Scenario-A of RQ1 (see Table 3b).

6.2.2 Data Imbalance

Real world data often contains significantly more neutral examples than vulnerable ones. A model trained on such skewed dataset is susceptible to being considerably biased toward the majority class.

Approach. We compute percentage on vulnerable samples *w.r.t.* total number of samples from different datasets used in this paper as shown in Table 1.

Observations. We notice that several datasets exhibit a notable imbalance in the fraction of vulnerable and neutral examples; the percentage vulnerability is sometimes as low as 6 percent. The ratio of vulnerable and neutral examples varies depending on the project and the data collection strategy employed. Existing methods fail to adequately address the data imbalance during training. This causes two problems: (1) When pre-trained models are used (i.e., Scenario-A in RQ1) to predict vulnerabilities in the real world, the ratios of vulnerable and neutral examples differ significantly in training and testing datasets. This explains why pretrained models perform poorly (as seen in Table 3 b). (2) When the models are re-trained, they tend to be biased towards the class with the most examples (i.e., the majority class). This results in poor recall values (i.e., they miss a lot of true vulnerabilities) and hence, also the F1-score (as seen in Table 3c).

6.2.3 Learning Irrelevant Features

In order to choose a good DL model for vulnerability prediction, it is important to understand what features the model uses to make its predictions. A good model should assign greater importance to the vulnerability related code features.

Approach. To understand what features a model uses for its prediction, we find the feature importance assigned to the predicted code by the existing approaches. For token-based models such as VulDeePecker, SySeVR, and Russell *et al.*, we use Lemna to identify feature importance [9]. Lemna assigns each token in the input with a value ω_i^t, representing the contribution of that token for prediction. A higher value of ω_i^t indicates a larger contribution of token towards the prediction and vice versa. For graph-based models, such as Devign, Lemna is not applicable [9]. In this case, we use the activation value of each vertex in the graph to obtain the feature importance. The larger the activation, the more critical the vertex is.

Observations. To visualize the feature importances, we use a heatmap to highlight the most to least important segments of the code. Fig. 9 shows two examples of correct predictions. Fig. 9a shows an instance where Russell *et al.*'s token-based method accurately predicted a vulnerability. But, the features that were considered most important for the prediction (lines 2 and 3) are not related to the actual vulnerability that appears in buggy `sprintf` lines (lines 6, 8, and 10). We observe similar behavior in other token based methods.

```
1  link_layer_show(struct ib_port *p,
2          struct port_attribute *unused, char * buf){
3      switch (rdma_port_get_link_layer(
4                  p->ibdev, p->port_num)) {
5          case IB_LINK_LAYER_INFINIBAND:
6              return sprintf(buf, "%s\n", "InfiniBand");
7          case IB_LINK_LAYER_ETHERNET:
8              return sprintf(buf, "%s\n", "Ethernet");
9          default:
10             return sprintf(buf, "%s\n", "Unknown");
11     }
12 }
```

(a) Vulnerable code example in Draper [5] dataset correctly predicted by Russel *et al.*'s token-based method.

```
1   static int mov_read_dvc1(MOVContext *c,
2           AVIOContext *pb, MOVAtom atom) {
3       AVStream *st;
4       uint8_t profile_level;
5       if (c->fc->nb_streams < 1)
6           return 0;
7       st = c->fc->streams[c->fc->nb_streams-1];
8       if (atom.size >= (1<<28) || atom.size < 7)
9           return AVERROR_INVALIDDATA;
10      profile_level = avio_r8(pb);
11      if ((profile_level & 0xf0) != 0xc0)
12          return 0;
...     ...
18      st->codec->extradata_size = atom.size - 7;
19      avio_seek(pb, 6, SEEK_CUR);
20      avio_read(
21          pb, st->codec->extradata,
22              st->codec->extradata_size);
23      return 0;
24  }
```

(b) Vulnerable example from FFMPeg+Qemu [6] dataset correctly predicted by graph model. Other method could not predict the vulnerability in this example.

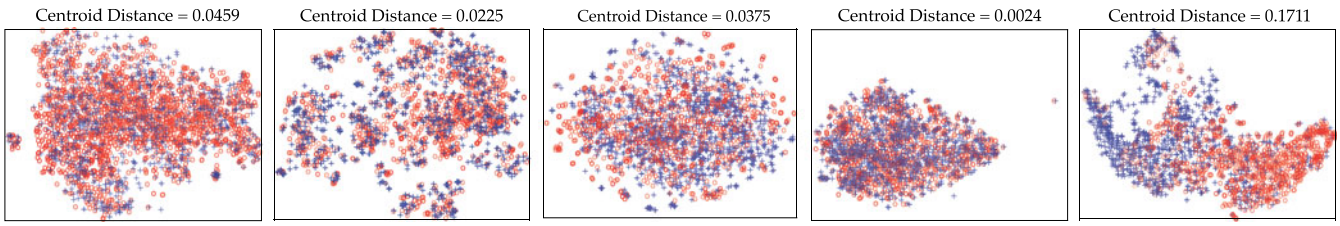
Fig. 9. Contribution of different code component in correct classification of vulnerability by different model. Underlined red shaded code elements are most contributing, *Italic green shaded* are the least. **Bold Blue** colored code are the source of vulnerabilities.

In contrast, Fig. 9b shows an example that was misclassified as neutral by token-based methods, but graph-based models accurately predict them as vulnerable. Here we note that the vulnerability is on line 20, and graph-based models use lines 3, 7, 19 to make the prediction, i.e. mark the corresponding function as vulnerable. We observe that each of these lines shares a data dependency with line 20 (through `pb` and `st`). Since graph-based models learn the semantic dependencies between each of the vertices in the graph through the code property graph, a series of connected vertices, each with high feature importance, causes the graph-based model to make the accurate prediction. Token-based models lack the requisite semantic information and therefore fail to make accurate predictions.

6.2.4 Model Selection: Lack of Class Separation

Existing approaches translate source code into a numeric feature vector that can be used to train a vulnerability prediction model. The efficacy of the vulnerability prediction model depends on how separable the feature vectors of the two classes (i.e., vulnerable examples and neutral examples) are. The greater the separability of the classes, the easier it is for a model to distinguish between them.

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.



(a) Russell *et al.* (b) VulDeePecker (c) SySeVR (d) Devign (e) REVEAL

Fig. 10. t-SNE plots illustrating the separation between vulnerable (denoted by +) and neutral (denoted by o) example. Existing methods fail to optimally separate vulnerable and neutral classes.

Approach. We use t-SNE plots to inspect the separability of the existing models. t-SNE is a popular dimensionality reduction technique that is particularly well suited for visualizing how high-dimensional datasets look in a feature space [41]. A clear separation in the t-SNE space indicates that the classes are distinguishable from one another. In order to numerically quantify the separability of the classes, we use the centroid distance proposed by Mao *et al.* [42]. We first find the centroids of each of the two classes. Next, we compute the euclidean distance between the the centroids. Models that have larger the euclidean distances are preferable since they exhibit greater class separation.

Observations. Fig. 10 illustrates the t-SNE plots of the existing approaches. All the existing approaches (Figs. 10a, 10b, 10c, and 10d) exhibit a significant degree of overlap in the feature space between the two classes. This is also reflected by the relatively low distance between the centroids in each of the existing methods. Among exiting methods, Devign (Fig. 10d) has the least centroid distance (around 0.0025); this is much lower than any other existing approach. This lack of separation explains why Devign, in spite of being a graph-based model, has poor real-world performance (see Table 3).

Result. Existing approaches have several limitations: they (a) introduce data duplication, (b) don't handle data imbalance, (c) don't learn semantic information, (d) lack class separability. DLVP may be improved by addressing these limitations.

6.3 How to Improve DLVP Approaches? (RQ3)

Motivation. To address challenges discussed in RQ2, we offer REVEAL— a road-map to help avoid some of the common problems that current state-of-the-art vulnerability prediction methods face when exposed to real-world datasets.

Approach. A detailed description of REVEAL pipeline is presented in Section 4. This pipeline offers the following benefits over the current state-of-the-art:

(1) *Addressing duplication:* REVEAL does not suffer from data duplication. During pre-processing, input samples are converted to their corresponding code property graphs whose vertices are embedded with a GGNN and aggregated with an aggregation function. This pre-processing approach tends to create a unique feature for every input samples. So long as the inputs are not exactly the same, the feature vector will also not be the same.

(2) *Addressing data imbalance:* REVEAL makes use of synthetic minority oversampling technique (SMOTE) to re-balance the distribution of vulnerable and neutral examples in the training data. This ensures that the trained model would be distribution agnostic and, therefore, better suited for real-world vulnerability prediction where the distribution of vulnerable and neutral examples is unknown.

(3) *Addressing model choice:* REVEAL extracts semantic as well as syntactic information from the source code using code property graphs. Using GGNN, each vertex embedding is updated with the embeddings of all its neighboring vertices. This further increases the semantic richness of the embeddings. This represents a considerable improvement to the current token-based and slicing-based models. As shown in Fig. 9b, REVEAL can accurately predict the vulnerability here.

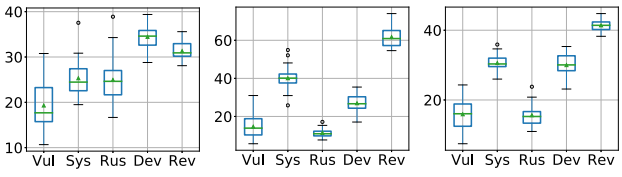
(4) *Addressing the lack of separability:* As shown in Figs. 10a, 10b, 10c, and 10d, the vulnerability class is almost inseparable from the non-vulnerability class in the feature space. To address this problem, REVEAL uses a representation learner that automatically learns how to re-balance the input feature vectors such that the vulnerable and neutral classes are maximally separated [11]. This offers significant improvements over the current state-of-the-art as shown in Fig. 10e. Compared to the other approaches of Figs. 10a, 10b, 10c, and 10d, REVEAL exhibits the highest separation between the vulnerable and neutral classes (roughly $85\times$ higher than other GGNN based vulnerability prediction).

We compare performance of REVEAL with existing vulnerability prediction approaches of two real-world datasets, i.e., FFMPeg+Qemu and REVEAL data.

Observations. Figs. 11 and 12 compare the performance of REVEAL tool with other approaches. We observe that REVEAL offers noticeable improvements in all the metrics:

- REVEAL*dataset:* REVEAL performs best in terms of F1-score and recall. The median recall is 60.91 percent (20.8 percent more than that of SySeVR, the next best model) and median F1-score is 41.25 percent (11.38 percent more than SySeVR). This represents a 51.85 and 36.36 percent improvement in recall and F1 over SySeVR respectively. While Devign (another GGNN based vulnerability prediction) produces a better precision, Devign's median recall 56.21 percent less than that of REVEAL. This indicates that, compared to Devign, REVEAL can find larger number of true-positive vulnerabilities (resulting in a better recall) at the cost slightly

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.



(a) Precision (b) Recall (c) F1-score

Fig. 11. Performance spectrum of ReVeal dataset. Vul = VulDeePecker [3], Sys=SySeVR [4], Rus = Russell *et al.* [5], Dev = Devign [6], Rev = ReVeal.

more false-positives (resulting in a slightly lower precision). Overall, ReVeal's median F1-score is 11.38 percent more than Devign, i.e., a 38.09 percent improvement.

- *FFMPeg+Qemu:* ReVeal outperforms other approaches in all performance metrics. ReVeal's median accuracy, precision, recall, and F1-scores are 5.01, 5.19, 13.11, and 12.64 percent higher respectively than the next best approach.

In the rest of this research question, we investigate contribution of each component of ReVeal.

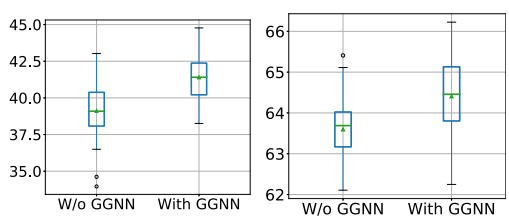
6.3.1 Contribution of Graph Neural Network

To understand the contribution of GGNN, we create a variant of ReVeal without GGNN. In this setup, we bypass the use GGNN and aggregate the initial vertex features to create the graph features. Further, we create another variant of ReVeal that uses *only GGNN without* re-sampling or representation learning.

Fig. 13 shows the F1-scores for the above setup. We observe that, in both ReVeal dataset and FFMPeg+Qemu, F1-score increases when we use GGNN in ReVeal's pipeline. We observe that the improvements offered by the use of GGNN is statistically significant (with a p-value of 0.0002 in ReVeal dataset, and 0.001 in FFMPeg+Qemu). Further, when we perform the A12 effect size [50] with 30 independent experiment runs in each case, we found that the the effect size is 81 percent for ReVeal dataset and 73 percent for FFMPeg+Qemu. This means that 81 percent of the times ReVeal performs better with GGNN than it does without GGNN in ReVeal dataset and 73 percent in FFMPeg +Qemu. Both of those effect sizes are considered large indicating ReVeal with GGNN is better than ReVeal without GGNN.

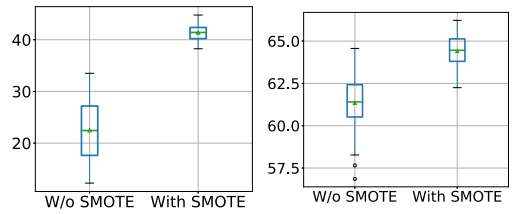
6.3.2 Effect of Training Data Balancing

To understand the contribution of SMOTE, we deploy two variants of ReVeal one with SMOTE and one without. Note that, ReVeal uses SMOTE as an off-the shelf data balancing tool. Choice of which data-balancing tool should be used is a configurable parameter in ReVeal's pipeline.



(a) ReVeal dataset (b) FFMPeg+Qemu

Fig. 12. Performance spectrum of FFMPeg+Qemu.



(a) ReVeal dataset (b) FFMPeg+Qemu

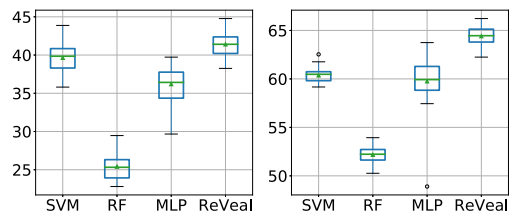
Fig. 13. Effect of GGNN in ReVeal's F1 score. The performance increase in both datasets when node information is propagated to the neighboring node through GGNN. The effect size is 0.81 (large) for ReVeal dataset and 0.73 for FFMPeg+Qemu.

Fig. 14 illustrates the effect of using data re-sampling in ReVeal's pipeline. We observe that re-balancing training data improves ReVeal's performance in general. The more skewed the dataset, the larger the improvement. In FFMPeg +Qemu, neutral examples populates roughly 55 percent of the data. There, using SMOTE offers only a 3 percent improvement in F1-score (see Fig. 14b). However, in ReVeal dataset, neutral examples populates 90 percent of the data, there we obtain more than 22 percent improvement in F1-score compared to not using SMOTE (see Fig. 14a). Without SMOTE, the precision of ReVeal tool improves and reaches up to 46.23 percent highest achieved precision among all the experimental settings. However, this setting suffers from low recall due to data imbalance. Thus, if an user cares more about precision over recall, SMOTE can be turned off, and vice versa.

6.3.3 Effect of Representation Learning

In order to understand the contribution of representation learning, we replace representation learning with three other learners: (a) Random Forest (a popular decision tree based classifier used by other vulnerability prediction approaches like Russell *et al.* [5]); (b) SVM with an RBF kernel which also attempts to maximize the margin between vulnerable and neutral instances [57]; and (c) An off-the-shelf Multi-Layer Perceptron.

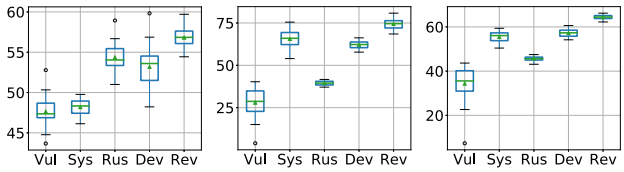
Fig. 15 shows the ReVeal's performance with different classification models. In both ReVeal dataset and FFMPeg +Qemu, our representation learner with triplet loss achieves the best performance. Max-margin models results in better performance in classifying vulnerable code in general. ReVeal with the representation learner performs statisti-



(a) ReVeal dataset (b) FFMPeg+Qemu

Fig. 14. Effect of training data re-balancing in ReVeal's performance (F1-score). In both datasets, re-balancing improves the performance of ReVeal.

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.



(a) Precision (b) Recall (c) F1-score

Fig. 15. ReVeal's performance (F1-score) in comparison to other machine learning models.

cally and significantly better than SVM in both ReVeal dataset and FFMPeg+Qemu (with $p - values < 0.01$ and $A12 > 0.6$). This is likely because SVM is a shallower than a representation learning model that propagates losses across several perceptron layers.

Result. The performance of DLVP approaches can be significantly improved using the ReVeal pipeline. The use of GGNN based feature embedding along with SMOTE and representation learning remedies data-duplication, data imbalance, and lack of separability. ReVeal produces improvements of up to 33.57 percent in precision and 128.38 percent in recall over state-of-the-art methods.

6.4 Results on Other Dataset

The results of ReVeal (and the effect of its constituent components) are shown in Table 5. In general, ReVeal vastly outperforms their models (see Table 3 a).

With respect to the Draper dataset, Russell *et al.* did release their entire dataset. Therefore, we were able to run ReVeal on their datasets. Due to the extreme imbalance in the Draper dataset (i.e., only $\approx 6\%$ are vulnerable examples), GGNN in ReVeal's pipeline was extremely biased towards the neutral examples. To solve this problem, we sub-sampled the neutral training examples to create a balanced dataset to train GGNN. We used this balanced dataset to train GGNN in ReVeal's pipeline. Table 6 shows the results of ReVeal in Draper dataset.

We observe a significant impact of training data rebalancing. In this experiment, we sub-sampled the training data for training GGNN. One might might wonder why use SMOTE in place of such simple under-sampling in other datasets. To this, we note that Draper is a large dataset (containing 1.2M examples in total). Thus, just with undersampling we are left with sufficiently large quantity of data to train a model. However, in ReVeal dataset and FFMPeg +Qemu, we need to increase number of vulnerable examples for training (i.e. we need to oversample). Thus, we first converted the code examples to feature space and used off-the-shelf SMOTE to re-balance the training data.

TABLE 5
Different Components of ReVeal's Performance in Juliet Dataset

Approach	Precision	Recall	F1
GGNN only	95.70	91.08	93.33
ReVeal-no-ggnn	94.89	90.27	92.52
ReVeal-no-smote	95.38	92.17	93.74
ReVeal-no-rl	94.15	95.89	95.01
ReVeal	94.94	96.38	95.65

TABLE 6
Different Component of ReVeal's Performance in Draper Dataset

Approach	Precision	Recall	F1
ReVeal-imbalance	10.28	91.24	18.47
GGNN only (balanced)	27.88	86.59	42.17
ReVeal-no-rl	29.57	88.34	44.30
ReVeal	36.24	87.29	51.22

7 Threats to Validity

7.1 Internal Validity

Tangled Commits. Tangled commits have long been studied in software engineering [58], [59] and a major setback for software evolution history driven research [60], [61]. Developers often combine more than one unrelated or weakly related changes in code in one commit [60] causing such a commit to be entanglement of more than one changes. Our collected ReVeal data is also subject to such a threat of containing tangled code changes. That said, to validate that the empirical finding in the paper are not biased by the tangled commits, we created an alternate version of ReVeal data, where we removed any patch that changes more than one function from consideration. In that version of ReVeal data, we find that ReVeal achieves 26.33 percent f1-score (compared to 41.25 percent f1-score in ReVeal's dataset). In contrast, if we do not use representation learning, ReVeal's f1 score drops to 22.95 percent. If we do not use the data balancing, ReVeal's performance drops to 13.13 percent. When we remove GGNN from ReVeal's pipeline, f1 score drops to 22.82 percent. These results corroborates the importance of GGNN, data balancing and representation learning in ReVeal's pipeline irrespective of existence of tangled code changes.

Multi-Function Vulnerabilities. Our formulation of VD is classification of a function. Some vulnerabilities might exhibit across multiple functions in the codebase. Annotating all such functions may hurt the reliability or applicability of a model, posing a threat to the validity of ReVeal. That said, we find that, in most cases (80 percent), vulnerability patches do not span more than 4 functions (see Fig. 6)

Another possible strategy to mitigate this threat is to consider file level vulnerability instead of functions within each file. However, we found that the code property graph (CPG) representation corresponding to a file becomes intractably large and current implementations of graph based neural networks are not adept at scaling very well *w.r.t.* such large graphs.

7.2 External Validity

Unknown Vulnerabilities. Existing DLVP models, including our pipeline, rely on inferring vulnerabilities based on past semantic and syntactic patterns. As a result, these models may not be effective in discovering vulnerability as yet unseen in past training data. For discovering such

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.

vulnerabilities, one may use other vulnerability detection paradigms such as fuzzing [62], [63], or a hybrid of static and dynamic analysis techniques [64].

Modularity of ReVeal*'s Pipeline.* ReVeal uses different components as modules in its pipeline. Each of the (trainable) modules are trained individually. While such modularity makes ReVeal very agile, it may miss some vulnerability patterns which might otherwise be found if trained jointly. We view this research as a roadmap towards building strong DLVP tool, thus modularity and agility are very important factors in this works so that users may optionally each component by their chosen alternatives, e.g., Representation learning may be replaced with SVM for training speed-up, or GGNN may be replaced with other unsupervised graph embedding approach such as node2vec or deepwalk.

8 Related Work

Vulnerability detection (vulnerability prediction) in software has been a significant research problem in software engineering. Traditional vulnerability detection systems fall into three major categories – static analysis [26], [65], dynamic analysis [66], [67], and symbolic analysis [68], [69]. Static analyzers analyze the static properties of the code i.e., AST [70], Flow/Dependency graphs [8], [71]. While static analyzers are often lightweight, used earlier in the development pipeline, such analyzers often cause a high number of false positives [72]. In contrast, dynamic analyzers execute the program and analyze for potential vulnerabilities of the program *w.r.t.* the runtime behavior of the program [67]. Dynamic analyzers detect vulnerabilities by reasoning about the user input (e.g., taint tracking) or generating useful program input to force the model crash (e.g., fuzzing). While dynamic analyzers execute the program on real inputs, symbolic analyzers analyze the program based on symbolic inputs [68], [73].

In recent years, Deep Learning based vulnerability detection shown very promising results in both static analysis [3], [4], [5], [6], [55], and dynamic analyses [62], [63], [74], symbolic executions [75]. Traditionally, static analysis based Vulnerability detection depends on the hard-coding vulnerability patterns. These patterns are learned from example data in DL-based vulnerability detection systems reducing developers' burden of hard-coding the patterns. Since Deep Learning is historically viewed as black-box in nature, a comprehensive study of such approaches in real world development is paramount. Also, given the widespread adoption of DL-based vulnerability prediction, it is distinctly important to analyze such research's feasibility in the real-world vulnerability prediction. Similar to this work, Li *et al.* [76] showed a comparative study of different DL based vulnerability prediction tools. However, their study did not compare against the real-world scenario. This work offers a detailed inspection of the existing researches. We throw light on some of the limitations of current vulnerability prediction approaches, and we propose a roadmap for collecting real world vulnerability data and improved modeling technique for real world vulnerability prediction.

9 Conclusion

In this paper, we systematically study different aspects of Deep Learning based Vulnerability Detection to effectively find real world vulnerabilities. We empirically show different shortcomings of existing datasets and models that potentially limits the usability of those techniques in practice. Our investigation found that existing datasets are too simple to represent real world vulnerabilities and existing modeling techniques do not completely address code semantics and data imbalance in vulnerability detection. Following these empirical findings, we propose a framework for collecting real world vulnerability dataset. We propose ReVeal as a configurable vulnerability prediction tool that addresses the concerns we discovered in existing systems and demonstrate its potential towards a better vulnerability prediction tool.

Acknowledgments

The authors would like to thank Yufan Zhuang for help in data collection and Dr. Suman Jana for their extensive feedback on this paper. This work was supported by National Science Foundation under Grants CCF 1845893, CCF 1822965, and CNS 1842456.

References

[1] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 672–681.

[2] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, "Questions developers ask while diagnosing potential security vulnerabilities with static analysis," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 248–259.

[3] Z. Li *et al.*, "VulDeepecker: A deep learning-based system for vulnerability detection," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[4] Z. Li *et al.*, "SySeVR: A framework for using deep learning to detect software vulnerabilities," 2018, *arXiv: 1807.06756*.

[5] R. Russell *et al.*, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl.*, 2018, pp. 757–762.

[6] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 10197–10207.

[7] E. Yudkowsky, "Artificial intelligence as a positive and negative factor in global risk," *Glob. Catastrophic Risks*, vol. 1, no. 303, 2008, Art. no. 184.

[8] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Secur. Privacy*, 2014, pp. 590–604.

[9] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "LEMNA: Explaining deep learning based security applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 364–379.

[10] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano, "Rusboost: A hybrid approach to alleviating class imbalance," *IEEE Trans. Syst., Man, Cybern.-Part A Syst. Hum.*, vol. 40, no. 1, pp. 185–197, 2009.

[11] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013.

[12] J. S. Bridle, "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition," in *Proc. Neurocomput.*, 1990, pp. 227–236.

[13] B. W. SUTER, "The multilayer perceptron as an approximation to a Bayes optimal discriminant function," *IEEE Trans. Neural Netw.*, vol. 1, no. 4, p. 296–298, Dec. 1990.

[14] K. Plunkett and J. L. Elman, *Exercises in Rethinking Innateness: A Handbook for Connectionist Simulations*. Cambridge, MA, USA: MIT Press, 1997.

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.

[15] V. Okun, A. Delaitre, and P. E. Black, "Report on the static analysis tool exposition (sate) IV," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Rep. 500–297, 2013.
[16] NIST, "Software assurance reference dataset," Oct. 2018. [Online]. Available: https://samate.nist.gov/SRD/index.php
[17] H. Booth, D. Rike, and G. Witte, "The national vulnerability database (NVD): Overview," Nat. Inst. Standards Technol., Tech. Rep., 2013.
[18] Cwe-761 example, VuldeePecker, 2017. [Online]. Available: https://github.com/CGCL-codes/VulDeePecker/blob/master/CWE-399/source_files/112611/CWE761_Free_Pointer_Not_at_Start_of_Buffer__char_console_81_bad.cp p
[19] P. Mackerras, "CVE-2020–8597 patch commit." [Online]. Available: https://github.com/paulusmack/ppp/commit/8d7970b8f3db727fe798b65f3377fe 6787575426
[20] Buffer overflow vulnerability in point-to-point protocol daemon (pppd). [Online]. Available: https://bit.ly/2XrQWc1
[21] CVE-2020–8597. [Online]. Available: https://security-tracker.debian.org/tracker/CVE-2020–8597
[22] A. Delaitre, V. Okun, and E. Fong, "Of massive static analysis data," in *Proc. IEEE 7th Int. Conf. Softw. Secur. Rel. Companion*, 2013, pp. 163–167.
[23] J. Turian, L. Ratinov, and Y. Bengio, "Word representations: A simple and general method for semi-supervised learning," in *Proc. 48th Annu. Meeting Assoc. Comput. Linguistics*, 2010, pp. 384–394.
[24] X. Rong, "Word2vec parameter learning explained," 2014, *arXiv:1411.2738*.
[25] CWE-761. [Online]. Available: https://cwe.mitre.org/data/definitions/761.html
[26] L. Developers, "Clang," Oct. 2019. [Online]. Available: clang.llvm.org
[27] G. Wu and E. Y. Chang, "Class-boundary alignment for imbalanced dataset learning," in *Proc. ICML Workshop Learn. Imbalanced Data sets II, Washington, DC*, 2003, pp. 49–56.
[28] A. Aggarwal and P. Jalote, "Integrating static and dynamic analysis for detecting vulnerabilities," in *Proc. 30th Annu. Int. Comput. Softw. Appl. Conf.*, vol. 1. 2006, pp. 343–350.
[29] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Inf. Softw. Technol.*, vol. 53, no. 4, pp. 363–387, 2011.
[30] I. Pashchenko, S. Dashevskyi, and F. Massacci, "Delta-bench: Differential benchmark for static analysis security testing tools," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2017, pp. 163–168.
[31] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 914–919.
[32] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," 2015, *arXiv:1511.05493*.
[33] R. Zhao, D. Wang, R. Yan, K. Mao, F. Shen, and J. Wang, "Machine health monitoring using local feature-based gated recurrent unit networks," *IEEE Trans. Ind. Electron.*, vol. 65, no. 2, pp. 1539–1548, Feb. 2018.
[34] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," 2017, *arXiv: 1711.00740*.
[35] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," 2020, *arXiv: 2006.03511*.
[36] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, 2002.
[37] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proc. IEEE/ACM 37th Int. Conf. Softw. Eng.*, vol. 2. 2015, pp. 99–108.
[38] L. Lusa *et al.*, "Smote for high-dimensional class-imbalanced data," *BMC Bioinf.*, vol. 14, no. 1, 2013, Art. no. 106.
[39] S. Barua, M. M. Islam, X. Yao, and K. Murase, "MWMOTE–Majority weighted minority oversampling technique for imbalanced data set learning," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 2, pp. 405–425, Feb. 2014.
[40] S. Barua, M. M. Islam, and K. Murase, "PROWSYN: Proximity weighted synthetic oversampling technique for imbalanced data set learning," in *Proc. Adv. Knowl. Discov. Data Mining*, J. Pei, V. S. Tseng, L. Cao, H. Motoda, and G. Xu, Eds., 2013, pp. 317–328.
[41] L. V. D. Maaten and G. Hinton, "Visualizing data using T-SNE," *J. Mach. Learn. Re.*, vol. 9, pp. 2579–2605, 2008.
[42] C. Mao, Z. Zhong, J. Yang, C. Vondrick, and B. Ray, "Metric learning for adversarial robustness," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 478–489.
[43] E. Hoffer and N. Ailon, "Deep metric learning using triplet network," in *Proc. Int. Workshop Similarity-Based Pattern Recognit.*, 2015, pp. 84–92.
[44] J. Wang, F. Zhou, S. Wen, X. Liu, and Y. Lin, "Deep metric learning with angular loss," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2017, pp. 2612–2620.
[45] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 815–823.
[46] F. Girosi, M. Jones, and T. Poggio, "Regularization theory and neural networks architectures," *Neural Comput.*, vol. 7, no. 2, pp. 219–269, Mar. 1995.
[47] D. M. Powers, "Evaluation: From precision, recall and F-measure to ROC, informedness, markedness & correlation," *J. Mach. Learn. Technol.*, vol. 2, pp. 37–63, 2013.
[48] H. D. Beale, H. B. Demuth, and M. Hagan, *Neural Network Design*. Boston, MA, USA: PWS Publishing, 1996.
[49] P. Koehn, "Statistical significance tests for machine translation evaluation," in *Proc. Conf. Empirical Methods Nat. Lang. Process.*, 2004, pp. 388–395.
[50] M. R. Hess and J. D. Kromrey, "Robust confidence intervals for effect sizes: A comparative study of COHEN's D and CLIFF's delta under non-normality and heterogeneous variances," in *Proc. Annu. Meeting Amer. Educ. Res. Assoc.*, 2004, pp. 1–30.
[51] A. Agrawal and T. Menzies, "Is "better data" better than "better data miners"?" in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2018, pp. 1050–1061.
[52] E. G. Jelihovschi, J. C. Faria, and I. B. Allaman, "Scottknott: A package for performing the scott-knott clustering algorithm in R," *TEMA (São Carlos)*, vol. 15, no. 1, pp. 3–17, 2014.
[53] R. W. Johnson, "An introduction to the bootstrap," *Teach. Statist.*, vol. 23, no. 2, pp. 49–54, 2001.
[54] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, p. 1–10.
[55] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVuldeePecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Trans. Dependable Secure Comput.*, early access, Sep. 23, 2019, doi: 10.1109/TDSC.2019.2942930.
[56] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proc. ACM SIGPLAN Int. Symp. New Ideas, New Paradigms, Reflections Program. Softw.*, 2019, pp. 143–153.
[57] B. Baesens *et al.*, "An empirical assessment of kernel type performance for least squares support vector machine classifiers," in *Proc. 4th Int. Conf. Knowl.-Based Intell. Eng. Syst. Allied Technol.*, vol. 1, 2000, pp. 313–316.
[58] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol. Reeng.*, 2015, pp. 341–350.
[59] S. Sothornprapakorn, S. Hayashi, and M. Saeki, "Visualizing a tangled change for supporting its decomposition and commit construction," in *Proc. IEEE 42nd Annu. Comput. Softw. Appl. Conf.*, vol. 1, 2018, pp. 74–79.
[60] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proc. 10th Working Conf. Mining Softw. Repositories* 2013, pp. 121–130.
[61] K. Herzig, S. Just, and A. Zeller, "The impact of tangled code changes on defect prediction models," *Empirical Softw. Eng.*, vol. 21, no. 2, pp. 303–336, 2016.
[62] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "MTFuzz: Fuzzing with a multi-task neural network," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 737–749.
[63] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *Proc. IEEE Symp. Secur. Privacy* 2019, pp. 803–817.
[64] M. Eskandari, Z. Khorshidpour, and S. Hashemi, "HDM-analyser: A hybrid analysis approach based on data mining techniques for malware detection," *J. Comput. Virol. Hacking Techn.*, vol. 9, no. 2, pp. 77–93, 2013.
[65] Flawfinder. [Online]. Available: https://www.dwheeler.com/flawfinder/
[66] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 475–485.

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.

[67] T. Wang, T. Wei, Z. Lin, and W. Zou, "IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2009, pp. 19–35.
[68] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 263–272, 2005.
[69] H. Li, T. Kim, M. Bat-Erdene, and H. Lee, "Software vulnerability detection using backward trace analysis and symbolic execution," in *Proc. Int. Conf. Availability, Rel. Secur.*, 2013, pp. 446–454.
[70] R. Ma, Z. Jian, G. Chen, K. Ma, and Y. Chen, "Rejection: A AST-based reentrancy vulnerability detection method," in *Proc. Chin. Conf. Trusted Comput. Inf. Secur.*, 2019, pp. 58–71.
[71] T. Wüchner, M. Ochoa, and A. Pretschner, "Robust and effective malware detection through quantitative data flow graph metrics," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2015, pp. 98–118.
[72] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proc. 7th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, 2007, pp. 1–8.
[73] C. Cadar, D. Dunbar, D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Oper. Syst. Des. Implementation*, vol. 8, 2008, pp. 209–224.
[74] D. She, Y. Chen, A. Shah, B. Ray, and S. Jana, "Neutaint: Efficient dynamic taint analysis with neural networks," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1527–1543.
[75] S. Shen, S. Shinde, S. Ramesh, A. Roychoudhury, and P. Saxena, "Neuro-symbolic execution: Augmenting symbolic execution with neural constraints," in *Proc. Netw. Distrib. Syst. Secur. Symp*, 2019, pp. 1–15.
[76] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, "A comparative study of deep learning-based vulnerability detection system," *IEEE Access*, vol. 7, pp. 103184–103197, 2019.

Saikat Chakraborty received the BSc degree from the Bangladesh University of Engineering and Technology. He is currently working toward the PhD degree with Columbia University, New York. His research interests include machine learning for reliable softwares and combining traditional program analysis techniques with robust machine learning to develop better bug detection and program repair tool.

Rahul Krishna received the PhD degree from NC State University. He is currently a research staff member with IBM Research, Yorktown Heights. His current research interests include the interaction between program analysis and machine learning to migrate apps to the hybrid cloud, artificial intelligence, configuration optimization, and security.

Yangruibo Ding received the BE degree in software engineering from the University of Electronic Science and Technology of China and the MS degree in computer science from Columbia University, New York, where he is currently working toward the PhD degree. His research interests include machine learning for program analysis and automated software engineering.

Baishakhi Ray received the PhD degree from the University of Texas, Austin. She is currently an assistant professor with the Department of Computer Science, Columbia University, NY, USA. She has authored or coauthored in CACM research highlights and has been widely covered in trade media. Her research focuses on the intersection of software engineering and machine learning. She was the recipient of best paper awards at the FASE 2020, the FSE 2017, the MSR 2017, the IEEE Symposium on Security and Privacy, Oakland, 2014, the NSF CAREER Award, the VMware Early Career Faculty Award, and the IBM Faculty Award.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on December 12,2022 at 01:47:14 UTC from IEEE Xplore. Restrictions apply.