Deep Learning based Vulnerability Detection: Are We There Yet?

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, 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% 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%. 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% boost in precision and 128.38% 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. In that spirit, we make available all the artifacts supporting our results: https://git.io/J6IA

Index Terms—Software Vulnerability, Deep Learning, Graph Neural Network.

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]-[5]. For example, static analysis-based tools typically result in high false positives, i.e., detect non-vulnerable cases as vulnerable, and dynamic analysis suffers from high false negatives, i.e., cannot detect many real vulnerabilities. After prolonged effort, these tools remain unreliable, leaving significant manual 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. In fact, several recent studies have demonstrated very promising results achieving high accuracy (up to 95%) at detecting vulnerabilities [6]-[12].

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 most importantly, whether they can be used effectively and reliably in detecting real-world vulnerabilities. Understanding such explainability and generalizability of the DL models is pertinent as it may help solve similar problems in other domains like computer vision [13], [14].

For instance, the generalizability of a DL model is 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. [15] 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 [6]-[8], [12] that have been reported to detect security vulnerabilities with high accuracy (up to 95%) in the existing literature. We primarily focus on the Deep Neural Network (DNN) models that take source code as input [6]-[8], [12], [15] and detect vulnerabilities at function granularity. These models operate on a wide range of datasets that are either generated synthetically or adapted from real-world code to fit in simplified vulnerability prediction settings.

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

---

1. published in TSE, ICSE, FSE, ASE, S&P Oakland, CCS, USENIX Security, etc.
samples are annotated as vulnerable/non-vulnerable, 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. [12].

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 ∼73%, on average. Even if we retrain these models with real-world data, their performance drops by ∼54% from the reported results. For example, VulDeePecker [6] reported a precision of 86.9% in their paper. However, when we use VulDeePecker’s pre-trained model in real world datasets, its precision reduced to 11.12%, and after retraining, the precision becomes 17.68%. A thorough investigation of such poor performance reveals several problems:

- **Inadequate Model.** The most popular models are token-based, which treat code as a sequence of tokens and do not take into account semantic dependencies that play a vital role in vulnerability predictions. Even when a graph-based model is used, it does not focus on increasing the class-separation between vulnerable and non-vulnerable categories. Thus, in realistic scenarios, they suffer from low precision and recall.

- **Learning Irrelevant Features.** While looking at the features that the existing techniques are picking up (using state-of-the-art explanation techniques [17], [18]), we find that the state-of-the-art models are essentially picking up irrelevant features that are not related to vulnerabilities and are likely artifacts of the training datasets.

- **Data Duplication.** The training and testing data in most existing approaches contain duplicates (up to 68%); thus, artificially inflating the reported results.

- **Data Imbalance.** Existing approaches do not alleviate the class imbalance problem [19], [20] of real-world vulnerability distribution as non-vulnerable code is much more frequent than the vulnerable ones.

Having established these concerns empirically, 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 [21] can be used on top of traditional DL methods to increase the class separation between vulnerable and non-vulnerable samples. Representation learning is a popular class of machine learning techniques that automatically discovers the input representations needed for improving classification, and thus, replaces the need for manual feature engineering. Our key insight is as follows: distinguishing features of vulnerable and benign code is complex; thus, the model must learn to represent them automatically in the feature space.

We further empirically establish that using semantic information (with graph-based models), data de-duplication, and balancing training data to address the class imbalance of vulnerable/non-vulnerable samples 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% respectively over current baselines.

In summary, our contributions in this paper are:

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. We release our dataset in this anonymous directory [https://bit.ly/3Bx30ai](https://bit.ly/3Bx30ai).

To this end, we argue 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.

## 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 of the model is evaluated on unseen test examples.

This section describes the theory of DL-based vulnerability prediction approaches ([2.1]), existing datasets ([2.2]), existing modeling techniques ([2.3]), and evaluation procedure ([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 non-vulnerable. Given a code element ($x$) and corresponding vulnerable/non-vulnerable label ($y$), the goal of the model is to learn features that maximize the probability $p(y|x)$ with respect to the model parameters ($\theta$). Formally, training a model is learning the optimal parameter settings ($\theta^*$) such that,

$$\theta^* = \text{argmax}_\theta \prod_{(x,y) \in D_{train}} p(y|x, \theta)$$  \hspace{1cm} (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. In general, this transformation and probability calculation is achieved through a feed forward layer and a softmax layer in the model. Typically, for binary classification tasks like vulnerability prediction, optimal model parameters are learned by minimizing the cross-entropy loss \( [23] \). Cross-entropy loss penalizes the discrepancy in the model’s predicted probability and the actual probability (0. for non-vulnerable 1. for vulnerable examples) \( [24] \). For instance, Zhou et al. \( [8] \), contains functions that are collected from open-source repositories but are annotated using static analyzers. Examples of SARD \( [26] \) and National Vulnerability Database (NVD \( [27] \)) 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.

2.2 Existing Dataset

To train a vulnerability prediction model, we need a set of annotated code with labels vulnerable or benign. The number of vulnerable code should be large enough to allow the model to learn from it. Researchers used a wide spectrum of data sources to collect data for DLVP (see Figure 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 \( [25] \) dataset and SARD \( [26] \) fall in this category. Here the examples are synthesized using known vulnerable patterns. These datasets were originally designed for evaluating traditional 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. \( [6] \), contains functions that are collected from open source repositories but are annotated using static analyzers. Examples of SARD \( [26] \) and National Vulnerability Database (NVD \( [27] \)) 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. \( [12] \) curated Devign dataset, which consists of past vulnerabilities and their fixes from four open-source projects, two of which are publicly available.

![Figure 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.](image)

![Figure 2: Example Vulnerability (CWE761) \( [28] \).](image)

![Figure 3: CVE-2020-8597 - A partial patch (original patch \( [29] \)) for an instance of buffer overflow vulnerability in linux point to point protocol daemon (pppd) due to a logic flaw in the packet processor \( [30], [31] \).](image)

**Limitations.** The problems with the dataset lie in how realistic the data source is and how they are annotated (see Figure 1). A model trained on a synthetic dataset comprising of simple patterns will be limited to detecting only those simple pattern which seldom occur in real life. For instance, consider an atypical buffer overflow example in Figure 2 used by VulDeePecker and SySeVR. Albeit a good pedagogical example, real world vulnerabilities are not as simple or as isolated. Figure 3 shows another buffer overflow example from linux kernel. Though the fix is very simple, finding the vulnerability itself requires an in-depth reasoning about the semantics of different components (i.e., variables, functions etc.) of the code. A model is trained to reason about simpler examples as in Figure 2 will fail to reason about Figure 3 code. Further, any model annotated by a static analyzer \( [8] \) inherits all the drawbacks, e.g., high false positive rate \( [1], [2] \). In the most realistic dataset, FFMpeg+Qemu \( [12] \), the ratio of vulnerable and non-vulnerable examples is approximately 45%-55%, which does not reflect the real world distribution of vulnerable code. Further, the dataset only contains function that annotates functions that went through vulnerability-fix commits as vulnerable. When a model is trained on such dataset, the model is not presented with other functions from a vulnerable functions’ context, thus will not be as effective in differentiating vulnerable functions from other non-vulnerable functions from the context.
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 [6, 8, 12].

- **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. [6] proposed a Bidirectional Long Short Term Memory (BS-LSTM) based model, Russell et al. [8] 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 [6] and SySeVR [7] 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 [12] leverage code property graph (CPG) proposed by Yamaguchi et al. [16] 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 [6] for details).

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 [6, 7]. For instance, VulDeePecker [6] replaces most of the variable and function names with symbolic names (VAR1, VAR2, FUNC1, FUNC2 etc.).

Expected input for all the models are real valued vectors commonly known as embeddings. There are several ways to embed tokens to vectors. One such way is to use an embedding layer [32] that is jointly trained with the vulnerability prediction task [8]. Another option is to use external word embedding tool(e.g., Word2Vec [33]) to create vector representation of every token. VulDeePecker [6] and SySeVR [7] uses Word2Vec to transform their symbolic tokens into vectors. Devign [12], 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. [8] optimized their model using cross-entropy loss, Zhou et al. [12] used regularized cross entropy loss.

**Limitations.** 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 [35–37]. To incorporate some semantic information, VulDeePecker [6] and SySeVR [7] extracted program slices of a potentially interesting point. For example, consider the code in Figure 1. A slice wrt 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++ ) } for ( data ; * 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 [16]. 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.

One problem with the existing approaches is that although the trained models learn to discriminate vulnerable and non-vulnerable code samples, the training paradigm does not explicitly focus on increasing the separation between the vulnerable and non-vulnerable examples. Thus, with slight variations the classifications become brittle.

Another problem pertains to data imbalance [38] between vulnerable and benign code as the proportion of vulnerable examples in comparison to the non-vulnerable one in real world dataset is extremely low [8]. When a model is trained on such imbalanced dataset, models tend to be biased by the non-vulnerable 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.

Limitations. Although all the existing approaches report their performances using their own evaluation dataset, it does not give a comprehensive overview of the applicability of the model in the real-world. All we can learn from such intra-dataset evaluation is how well their approach fits their own dataset. Although there are some limited case studies on such models finding vulnerabilities in real-world projects, those 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 [39] and too much of any would hold the developer from using the model [40, 41].

3 ReVEAL Data Collection

To address the limitations with the existing data sets, 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 Bugzill2. For Linux Debian Kernel, we collected the issues from Debian security tracker. 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 by Zhou et al. [42], where they filter out commits that do not have security related keywords.

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’.

A contrived example of our data collection strategy is illustrated in Figure 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’.

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, by retaining the fixed variant of the vulnerable function, the DL model may learn the nature of patch. We make available our data collection framework and the curated vulnerability data for Chromium and Debian3 for broader dissemination.

4 ReVEAL Pipeline

In this section, we present a brief overview of the ReVEAL pipeline that aims to more accurately detect the presence of real-world vulnerabilities. Figure 5 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 [41]. 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 non-vulnerable examples (Algorithm 1). Algorithm 2 shows the full training procedure for ReVEAL. This algorithm expects Training data – a list of tuples, where each tuple contains a code (C) and corresponding vulnerability annotation (l).

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. Note that, the feature extraction scheme presented below represents the most commonly used series of steps for extracting features from a graph representation [12]. ReVEAL uses Algorithm 2 to extract graph embedding (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) [16]. The CPG is a particularly useful representation of the original

2. https://bugs.chromium.org/p/chromium/issues/list
3. https://security-tracker.debian.org/tracker/
4. Chromium and Debian dataset: https://bit.ly/3bX30ai
Table 1: Summary of DLVP datasets and approaches.

| Dataset          | Used By             | # Programs | % Vul* | Granularity | Model Type | Model  | Description                                                   |
|------------------|---------------------|------------|--------|-------------|------------|--------|---------------------------------------------------------------|
| SATE IV Juliet   | Russell et al. [8]  | 11,396     | 45.00  | Function    | Token      | CNN+RF | Synthetic code for testing static analyzers.                  |
| SARD [26]        | VulDeePecker        | 9,851      | 31     | Slice       | Token      | BLSTM  | Synthetic, academic, and production security flaws or vulnerabilities. |
| NVD [27]         | VulDeePecker        | 14,000     | 13.41  | Slice       | Token      | BGRU   | Collection of known vulnerabilities from real world projects.  |
| Draper [8]       | Russell et al. [8]  | 1,274,366  | 6.46   | Function    | Token      | CNN+RF | Contains code from public repositories in Github and Debian source repositories. |
| FFMPeg+Qemu [12] | Devign [12]         | 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.

1 VulDeePecker and SySeVR uses combination of SARD and NVD datasets to train and evaluate their model.

![Diagram](image)

Figure 6: Overview of the REVEAL vulnerability prediction framework.

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 [16].

Formally, a CPG is denoted as $G = (V, E)$, 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) [43] for this purpose.

Feature vectors for all the nodes in the graph ($X$) along with the edges ($E$) are the input to the GGNN [42, 43]. 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,

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

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 [43–45]. $x'_v$ 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.:

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

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 [12]. Such pre-training deconstructs the task of “learning code representation”, and “learning vulnerability”, and is also used by Russell et al. [8]. While, we pre-train GGNN in a supervised fashion, unsupervised program representation learning [46] can also be done to learn better program presentation. However, such learning
Algorithm 1: REVEAL.

Input: Train data – $D_{train}$, 
Contribution of triplet loss – $\alpha$, 
Contribution of regularization loss – $\beta$, 
Separation boundary – $\gamma$, 
Learning rate – $lr$

Output: Trained model.
1 Function REVEAL:
2 \hspace{1em} features $\leftarrow \emptyset$
3 \hspace{1em} labels $\leftarrow \emptyset$
4 \hspace{2em} $\triangleright$ Extract features from every code
5 \hspace{2em} for $(C, t) \in D_{train}$ do
6 \hspace{3em} $f \leftarrow$ embed_features($C$
7 \hspace{3em} features $\leftarrow$ features $\cup$ $f$
8 \hspace{3em} labels $\leftarrow$ labels $\cup t$
9 \hspace{2em} end
10 \hspace{2em} $\triangleright$ Rebalance with SMOTE.
11 \hspace{2em} $D_{balanced}$ $\leftarrow$ SMOTE(features, labels)
12 \hspace{2em} $M$ $\leftarrow$ RepresentationLearningModel()
13 \hspace{2em} $\triangleright$ Train Model.
14 \hspace{2.5em} for $(x_g, l_g)$ $\in D_{balanced}$ do
15 \hspace{3.5em} $\triangleright$ Define loss function.
16 \hspace{3.5em} $L_{all}$ $\leftarrow$ loss_function($M$, $D_{balanced}$, $x_g$, $l_g$, $\alpha$, $\beta$, $\gamma$)
17 \hspace{3.5em} $\triangleright$ $\theta$ represents the model parameters of $M$.
18 \hspace{3.5em} $\theta$ $\leftarrow$ $\theta$ $-$ $\nabla_{\theta} (L_{all})$
19 \hspace{2.5em} end
20 return $M_{\theta}$

Algorithm 2: Graph Embedding

Input: Code – $C$
Output: Feature vector $x_g$ representing $C$
1 Function embed_features($C$):
2 \hspace{1em} $(V, E)$ $\leftarrow$ extract_code_property_graph($C$
3 \hspace{1em} $X$ $\leftarrow$ $\emptyset$
4 \hspace{2em} for $v \in V$ do
5 \hspace{3em} $T_v$ $\leftarrow$ onehot($v$.type())
6 \hspace{3em} $C_v$ $\leftarrow$ word2vec($v$.code_fragment())
7 \hspace{3em} $x_v$ $\leftarrow$ $\text{concat}(T_v, C_v)$
8 \hspace{3em} $X$ $\leftarrow$ $x_v$ $\cup$ $X$
9 \hspace{2em} end
10 $X'$ $\leftarrow$ GGN($X$, $E$
11 $x_g$ $\leftarrow$ Aggregate($X'$
12 return $x_g$

Algorithm 3: Pseudocode for SMOTE.

Input: Training Dataset – $D_{train}$
Number of Nearest Neighbors – $k$
Expected Number of samples per class – $m$
Output: Sampled Dataset – $D_{sampled}$
1 Function SMOTE ($D_{train}$, $k$, $m$):
2 \hspace{1em} $D_{sampled}$ $\leftarrow$ $D_{train}$
3 \hspace{2em} while # of Majority examples $>$ $m$ do
4 \hspace{3em} $x$ $\leftarrow$ random majority class example from $D_{sampled}$
5 \hspace{3em} Remove $x$ from $D_{sampled}$
6 \hspace{2em} end
7 \hspace{2em} while # of Minority examples $<$ $m$ do
8 \hspace{3em} $x$ $\leftarrow$ random minority class example from $D_{sampled}$
9 \hspace{3em} neighbors $\leftarrow$ $k$ nearest minority neighbors of $x$
10 \hspace{3em} for $n \in$ neighbors do
11 \hspace{4em} $x_s$ $\leftarrow$ interpolate($x$, $n$)
12 \hspace{4em} Add $x_s$ to $D_{sampled}$
13 \hspace{2em} end
14 \hspace{2em} end
15 return $D_{sampled}$

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 non-vulnerable 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 non-vulnerable examples exhibit a significant overlap in the feature space. This makes it difficult to demarcate the vulnerable examples from the non-vulnerable 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 non-vulnerable 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 non-vulnerable examples.

4.2.1 Reducing Class Imbalance

In order to handle imbalance in the number of vulnerable and non-vulnerable classes, we use the “synthetic minority over-sampling technique” (for short, SMOTE) [47]. 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 [48]–[54].

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 non-vulnerable examples from the training set. This process is repeated until a balance is reached between the vulnerable and non-vulnerable examples. We present the pseudo-code of SMOTE in Algorithm 3.

4.2.2 Representation Learning Model

The graph embedding of the vulnerable and non-vulnerable code samples at the end of Phase-I tend to exhibit a high degree of overlap in feature space. This effect is illustrated by the t-SNE plot [55] of the feature space in Figure 8(a)–(d). In these examples, there are no clear distinctions between the vulnerable (denoted by +) and the non-vulnerable samples (denoted by o). This lack of separation makes it particularly difficult to train an ML model to learn the distinction between the vulnerable and the non-vulnerable 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 between vulnerable and non-vulnerable samples. For this, we use a multi-layer perceptron (MLP) [23], 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 $\theta$ (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 projects 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, $\hat{y} = \sigma (W \cdot h(x_g) + b)$. Where $\sigma$ 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 non-vulnerable examples in the latent space, we adopt the triplet loss $L_{trp}$ as our loss function. Triplet loss has been widely used in machine learning, specifically in representation learning, to create a maximal separation between classes [57, 58]. The triplet loss is comprised of three individual loss functions: (a) cross entropy loss ($L_{CE}$); (b) projection loss ($L_p$); and (c) regularization loss ($L_{reg}$). It is given by:

$$L_{trp} = L_{CE} + \alpha \cdot L_p + \beta \cdot L_{reg}$$  (2)

$\alpha$ and $\beta$ 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,

$$L_{CE} = - \sum \hat{y} \cdot \log(y) + (1 - \hat{y}) \cdot \log(1 - y)$$  (3)

Here, $y$ is the true label and $\hat{y}$ represents the predicted label.

The second component of the triplet loss is used to quantify how well the latent representation can separate the vulnerable and non-vulnerable 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 non-vulnerable 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 $L_p$ which is defined by,

$$L_p = \|D(h(x_g), h(x_{same})) - D(h(x_g), h(x_{diff}))\| + \gamma$$  (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 than that of $x_g$. Further, $\gamma$ is a hyperparameter used to define a minimum separation boundary. Lastly, $D(\cdot)$ represents the cosine distance between two vectors and is given by,

$$D(v_1, v_2) = 1 - \frac{v_1 \cdot v_2}{\|v_1\| \cdot \|v_2\|}$$  (5)

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

The final component of the triplet loss is the regularization loss ($L_{reg}$) 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 [56, 59].

Such arbitrary increase in $h(x_g)$ prevents the model from converging [60, 61]. Therefore, we use a regularization loss ($L_{reg}$) to penalize latent representations ($h(x_g)$) that are larger in magnitude. The regularization loss is given by:

$$L_{reg} = ||h(x_g)|| + ||h(x_{same})|| + ||h(x_{diff})||$$  (6)

With the triplet loss function, REVEAL trains the model to optimize for it parameters (i.e., $\theta, W, b$) by minimizing equation $2$. The effect of using representation learning can be observed by the better separability of the vulnerable and non-vulnerable examples in Figure 8(b). Algorithm 4 shows the detailed algorithm for calculating the loss.

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 available. 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. Hyper-parameters for different components in REVEAL are shown in Table 2.

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 [6], SySeVR [7], Russell et al. [8], and Devign [12]) and REVEAL’s performance on two real world datasets (i.e., REVEAL dataset, and FFMPeg+Gemu). FFMPeg+Gemu was shared by Zhou et al. [12] 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
To understand a model’s performance, researchers and model developers need to understand the performance of a model against a known set of examples. There are two important aspects to note here, (a) the evaluation metric, and (b) the evaluation procedure.

**Problem Formulation and Evaluation Metric:** Most of the approaches formulate the problem as a classification 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 non-vulnerable) 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. [8], and Devgn [42], where we classify the function. This is the most suitable model working with the graph, since slices are paths in the graph.

We study approaches based on four popular evaluation metrics for classification task [62] – 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 [63], 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 80%, 10%, and 20% 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 [51], [64], [65] and effect size test [66]. 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 [67]–[71], we use a non-parametric bootstrap hypothesis test [72] in conjunction with the A12 effect size test [73]–[75]. We distinguish results from different experiments if both significance test and effect size test agreed that the division was statistically significant (99% confidence) and is not result of a “small” effect (A12 ≥ 60%) (similar to Agrawal et al. [51]).

6 **Empirical Results**

We present our empirical results as answers to the following research questions:

- **RQ1:** How effective are existing approaches for real-world vulnerability prediction? (6.1)
- **RQ2:** What are the limitations of existing approaches? (6.2)
- **RQ3:** How to improve DLVP approaches? (6.3)

### 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., 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 3a 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
Table 3: Performance of existing approaches in predicting real world vulnerability. All the numbers are reported as Median (IQR) format.

(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| Juliet         |                | 95.90| 82.50|        | 85.40|
|         | NVD/SARD       |                | 95.90| 82.50|        | 85.20|
|         | FFMPeg+Qemu    |                | 72.26|      |        | 73.26|
|         | Russell et al. | Juliet         | 38.11| 41.36| 6.51   | 11.24|
|         |                | Draper         | 70.08| 49.05| 15.61  | 23.66|
|         |                | Devign         | 66.24| 10.74| 37.04  | 16.66|

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

| Dataset | Technique      | Training       | Acc  | Prec | Recall | F1   |
|---------|----------------|----------------|------|------|--------|------|
| REVEAL  | VulDeePecker   | NVD/SARD       | 79.05| 11.12| 13.64  | 12.18|
|         | SySeVR         | NVD/SARD       | 79.48| 9.38 | 15.89  | 10.37|
|         | Russell et al. | Juliet         | 38.11| 41.36| 6.51   | 11.24|
|         |                | Draper         | 70.08| 49.05| 15.61  | 23.66|
|         |                | Devign         | 66.24| 10.74| 37.04  | 16.66|

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

| Dataset | Input          | Approach       | Acc  | Prec | Recall | F1   |
|---------|----------------|----------------|------|------|--------|------|
| REVEAL  | Token          | Russell et al. | 90.98| 24.63| 10.91  | 15.24|
|         | Slice + Token  | VulDeePecker   | 89.05| 17.68| 13.87  | 15.7 |
|         |                | SySeVR         | 84.22| 24.46| 40.11  | 30.25|
|         | Graph          | Devign         | 88.41| 34.61| 26.67  | 29.87|
| FMPEG+QEMU| Token         | Russell et al.| 58.13| 54.04| 39.50  | 45.62|
|         | Slice + Token  | VulDeePecker   | 53.58| 47.36| 28.70  | 35.20|
|         |                | SySeVR         | 52.52| 48.34| 65.96  | 56.03|
|         | Graph          | Devign         | 58.57| 53.60| 62.73  | 57.18|

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 ~94% from the reported results.

| 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  | VulDeePecker            | 25.85           |
|         | SySeVR                  | 25.56           |
|         | Russell et al.           | 8.93            |
| FMPEG+QEMU | VulDeePecker  | 19.58           |
|         | SySeVR                  | 22.10           |
|         | Russell et al.           | 20.54           |

Table 4: Percentage of duplicate samples in datasets.

6.2 Key limitations of existing DLVP approaches (RQ2)

Motivation. In RQ1, we showed that existing approaches are not effective in detecting real-world vulnerabilities. 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.

Approach. We apply each preprocessing technique to its respective dataset (see Table 3a) 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%, and FFMpeg+Qemu had 0.2%). After preprocessing, although some duplicates are introduced (e.g., SySeVR’s preprocessing technique introduces 25.56% duplicates in REVEAL dataset and 22.10% 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 \cite{7, 76}, 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% examples are unique. Duplication tends to artificially inflate the overall performance of a method \cite{77}, 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 non-vulnerable 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 non-vulnerable examples; the percentage vulnerability is sometimes as low as 6%. The ratio of vulnerable and non-vulnerable 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 non-vulnerable examples differ significantly in training and testing datasets. This explains why pretrained models perform poorly (as seen in Table 3b). (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 Lemma to identify feature importance \cite{18}. Lemma assigns each token in the input with a value $\omega_i$, representing the contribution of that token for prediction. A higher value of $\omega_i$ indicates a larger contribution of token towards the prediction and vice versa. For graph-based models, such as Devign, Lemma is not applicable \cite{15}. 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. Figure 7 shows two examples of correct predictions. Figure 7a 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.

In contrast, Figure 7b shows an example that was misclassified as non-vulnerable 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

```c
1 static int mov_read_dvcl(MOVContext *c, AVIOContext *pb, MOVAtom *atom) {
2     AVStream *st = c->stream_get_stream(c, atom->stream_idx);
3     if (st == NULL) {
4         return AVERROR_INVALIDDATA;
5     }
6     return AVERROR_INVALIDDATA;
7     return st->codec->extradata_size;
8     if (atom->size > = (1<<28) || atom->size <= 0)
9         return AVERROR_INVALIDDATA;
10     if ((profile_level & 0xc0) != 0)
11         return 0;
12     return 18;
13 }
14
15 (a) Vulnerable code example in Draper \cite{8} dataset correctly predicted by Russell et al.’s token-based method.
16
17 (b) Vulnerable example from FFMpeg+Qemu \cite{12} dataset correctly predicted by graph model. Other method could not predict the vulnerability in this example.
```

Figure 7: Contribution of different code component in correct classification of vulnerability by different model. Red-shaded code elements are most contributing, Green-shaded are the least. Red colored code are the source of vulnerabilities.
Figure 8: t-SNE plots illustrating the separation between vulnerable (denoted by +) and non-vulnerable (denoted by o) example. Existing methods fail to optimally separate vulnerable and non-vulnerable classes.

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 non-vulnerable examples) are. The greater the separability of the classes, the easier it is for a model to distinguish between them.

**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 [55]. 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. [56]. We first find the centroids of each of the two classes. Next, we compute the euclidean distance between the centroids. Models that have larger the euclidean distances are preferable since they exhibit greater class separation.

**Observations.** Figure 8 illustrates the t-SNE plots of the existing approaches. All the existing approaches (Figure 8a–e) 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 existing methods, Devign (Figure 8d) has the least centroid distance (around 0.025); 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.** In RQ2, we highlighted a number of challenges that limit the performance of existing DLVP on real-world datasets. To address these challenges, we offer ReVEAL—a roadmap 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 is presented in [84]. Briefly, it works as follows: (i) input code fragment is converted to a feature vector with the help of a code property graph and GGNN (§4.1); (ii) the feature vectors are resampled using SMOTE (§4.2.1) that addresses potential data imbalance; and finally, (iii) a multi layer perceptron based representation learner is trained to learn a representation of the feature vectors that maximally separates the positive and negative classes (§4.2.2). 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 non-vulnerable 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 non-vulnerable 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 Figure 8b, ReVEAL can accurately predict the vulnerability here.

4) **Addressing the lack of separability:** As shown in Figure 8a–e, 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 non-vulnerable classes are maximally separated [21]. This offers significant improvements over the current state-of-the-art as shown in Figure 8e. Compared to the other approaches of Figure 8a–d, ReVEAL exhibits the highest separation between the
vulnerable and non-vulnerable 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.** Figures 9 and 10 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% (20.8% more than that of SySeVR, the best model) and median F1-score is 41.25% (11.38% more than SySeVR). This represents a 51.85% and 36.36% 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% 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 more false-positives (resulting in a slightly lower precision). Overall, REVEAL’s median F1-score is 11.38% more than Devign, i.e., a 38.09% 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% higher respectively than the next best approach.

In the rest of this research question, we investigate contribution of each component of REVEAL. Specifically, we study what improvements are offered by the use of (a) Graph neural network (6.3.1); (b) re-balancing training data with (6.3.2); and finally (c) representation learning (6.3.3).

### 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.

Figure 11 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 [66] with 30 independent experiment runs in each case, we found that
Table 6: Impact of re-balancing training data in REVEAL’s performance (Median/IQR). FFMPeg+Qemu is almost balanced, thus further re-balancing do not impact performance very much. However, in REVEAL dataset, training data re-balancing improved the performance significantly.

| Dataset       | Approach | Accuracy | Precision | Recall | F1-score |
|---------------|----------|----------|-----------|--------|----------|
| REVEAL        | W/O Re-balance | 90.48 (0.93) | 46.23 (11.30) | 15.09 (9.09) | 22.44 (9.56) |
|               | With Re-balance | 84.37 (1.73) | 30.91 (2.76) | 60.91 (7.89) | 41.25 (2.28) |
| FFMPeg+Qemu   | W/O Re-balance | 62.94 (1.40) | 58.86 (2.88) | 64.20 (6.76) | 61.40 (1.91) |
|               | With Re-balance | 62.51 (0.90) | 56.85 (1.54) | 74.61 (4.31) | 64.42 (1.33) |

the the effect size is 81% for REVEAL dataset and 73% for FFMPeg+Qemu. This means that 81% of the times REVEAL performs better with GGNN than it does without GGNN in REVEAL dataset and 73% in FFMPeg+Qemu. Both of those effect sizes are considered large indicating REVEAL with GGNNN’s f1-score distribution is better than REVEAL without GGNNN.

We contend that, since GGNN embeds the neighbors’ information in every vertex, vertices have richer information about the graph. Thus REVEAL’s classification model have more information at its disposal to reason about. The result indicates that when vertices assimilate information from neighboring vertices, vulnerability prediction performance increases.

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.

Figure 13 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, non-vulnerable examples populates roughly 55% of the data. There, using SMOTE offers only a 3% improvement in F1-score (see Figure 12a). However, in REVEAL dataset, non-vulnerable examples populates 90% of the data, there we obtain more than 22% improvement in F1-score compared to not using SMOTE (see Figure 12c). Without SMOTE, the precision of REVEAL tool improves and reaches up to 46.23% (see Table 7 in Appendix), 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. [8]); (b) SVM with an RBF kernel which also attempts to maximize the margin between vulnerable and non-vulnerable instances [78]; and (c) An off-the-shelf Multi-Layer Perceptron which uses a log-Likelihood loss [79].

Figure 13 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. REVEAL’s median F1-score is 62.8%, 13.3%, 3.5% higher than that of RF, MLP, and SVM baselines respectively in REVEAL dataset. For FFMPeg+Qemu improvement in median F1-score is 23.33%, 7.5%, 6.5% over RF, MLP, and SVM respectively.

Max-margin models results in better performance in classifying vulnerable code in general. REVEAL with the representation learner performs statistically 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 GNNN 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% in precision and 128.38% in recall over state-of-the-art methods.

7 DISCUSSION
7.1 Vulnerability Detection in Real World
The usefulness of a source code vulnerability detection tool depends on its use case scenario. Ideally, in a real-world scenario, developers would deploy a trained vulnerability prediction model to identify vulnerable functions from a codebase. In simple terms, given all the functions in the code base, developers would want to locate the vulnerable function. As discussed in Section 5 evaluating such a scenario is paramount in understanding the usefulness of an approach. Existing approaches show very little to no
evaluation of how respective approaches perform in such a real-world scenario. For instance, Devign [12] showed a simulated real imbalanced evaluation (we refer to Table 3 in Devign’s original paper). However, for that simulation, they randomly sampled their version of the test data to contain 10% vulnerable example. Their reported results also show the drop in performance in real world imbalanced settings. Nevertheless, we hypothesize that their evaluation method does not truly reflect how a method will perform in the real world since the imbalance in their dataset is artificial. In this study, we propose a method to simulate such an evaluation scenario. Therefore, we hope such evaluation settings help drive new research in vulnerability detection.

7.2 Vulnerability Data and Tangled commits

Tangled commits have long been studied in software engineering [82], [83] and a major setback for software evolution history driven research [84]–[86]. Developers often combine more than one unrelated or weakly related changes in code in one commit [84] 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. Thus, we investigate the characteristics of number of function changes in vulnerability-fix patches. Figure 14 shows a histogram of number of changed function in Vulnerability-fix patches. Percentage of changes account for 12 or fewer number of function changes.

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% f1 score. In contrast, if we do not use representation learning, REVEAL’s f1 score drops to 22.95%. If we do not use the data balancing, REVEAL’s performance drops to 13.13%. When we remove GGNN from REVEAL’s pipeline, f1 score drops to 22.82%. These results corroborates the importance of GGNN, data balancing and representation learning in REVEAL’s pipeline irrespective of existence of tangled code changes.

8 RELATED WORK

There have been a wide array of ML-based vulnerability prediction research [87]–[91]. Yamaguchi et al. [92] applied anomaly detection techniques on embeddings produced from static tainting to discover missing conditions such as input validation. Perl et al. [93] used commit messages to detect the vulnerabilities of a program. These work leveraged Support Vector Machine (SVM) for VP. Li et al. [94] used multi-class SVM to detect different class of vulnerabilities. Recently, DLVP has been subject to much research in both static- [8] and dynamic [95] analysis scenario. However, static settings are more popular [96] using code slices [6], trees [97], graphs [12] etc. Although most prominent approaches [6]–[8], [76] use token based representation of code, recent graph based modeling showed success in vulnerability prediction [12].

Code Property Graph (CPG), introduced by Yamaguchi et al. [16], models the combined semantic and syntactic information of a program. The CPG is a joint data structure that leverages the information from abstract syntax trees, control flow graphs and program dependency graphs. CPG has shown to be robust in reasoning about vulnerabilities [12], [16], [98]. Thus, REVEAL uses CPG to extract graph based features. In addition, REVEAL reduces data imbalance (through re-sampling in feature space) and learns to maximize separation between vulnerable and non-vulnerable examples.

There are several choices of techniques for reducing data imbalance [47], [49], [49] all of which use different strategies for balancing any imbalanced datasets. We choose SMOTE in REVEAL’s pipeline as it has shown to be successful in other software engineering related tasks [51], [100]–[103].

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.

ACKNOWLEDGEMENTS

We would like to thank Yufan Zhuang for initial help in data collection. We also thank Dr. Suman Jana for their extensive feedback on this paper.

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 Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013, pp. 672–681.
[92] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, “Chucky: Exposing missing checks in source code for vulnerability discovery,” in Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013, pp. 499–510.

[93] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, “Vcfinder: Finding potential vulnerabilities in open-source projects to assist code audits,” in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015, pp. 426–437.

[94] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, “Vulpecker: an automated vulnerability detection system based on code similarity analysis,” in Proceedings of the 32nd Annual Conference on Computer Security Applications. ACM, 2016, pp. 201–213.

[95] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, “Neuzz: Efficient fuzzing with neural program smoothing,” in 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019, pp. 803–817.

[96] D. Larochelle and D. Evans, “Statically detecting likely buffer overflow vulnerabilities,” in 10th USENIX Security Symposium, 2001.

[97] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, “Convolutional neural networks over tree structures for programming language processing,” in Proceedings of 30th AAAI Conference on Artificial Intelligence, 2016.

[98] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, “Automatic inference of search patterns for taint-style vulnerabilities,” in 2015 IEEE Symposium on Security and Privacy. IEEE, 2015, pp. 797–812.

[99] S. Barua, M. M. Islam, X. Yao, and K. Murase, “Mwmote–majority weighted minority oversampling technique for imbalanced data set learning,” IEEE Transactions on knowledge and data engineering, vol. 26, no. 2, pp. 405–425, 2012.

[100] H. Alsawalqah, H. Faris, I. Aljarah, L. Alnemer, and N. Alhindawi, “Hybrid smote-ensemble approach for software defect prediction,” in Computer Science On-line Conference. Springer, 2017, pp. 355–366.

[101] S. Guo, R. Chen, H. Li, T. Zhang, and Y. Liu, “Identify severity bug report with distribution imbalance by cr-smote and elm,” International Journal of Software Engineering and Knowledge Engineering, vol. 29, no. 02, pp. 139–175, 2019.

[102] C. Pak, T. T. Wang, and X. H. Su, “An empirical study on software defect prediction using over-sampling by smote,” International Journal of Software Engineering and Knowledge Engineering, vol. 28, no. 06, pp. 811–830, 2018.

[103] R. Pears, J. Finlay, and A. M. Connor, “Synthetic minority oversampling technique (smote) for predicting software build outcomes,” arXiv preprint arXiv:1407.2330, 2014.