An Empirical Study of the Imbalance Issue in Software Vulnerability Detection[*]

Yuejun Guo[1], Qiang Hu[2][**], Qiang Tang[1], and Yves Le Traon[2]

[1] ITIS, Luxembourg Institute of Science and Technology, Luxembourg
{yuejun.guo,qiang.tang}@list.lu
[2] SnT, University of Luxembourg, Luxembourg
{qiang.hu,yves.letraon}@uni.lu

Abstract. Vulnerability detection is crucial to protect software security. Nowadays, deep learning (DL) is the most promising technique to automate this detection task, leveraging its superior ability to extract patterns and representations within extensive code volumes. Despite its promise, DL-based vulnerability detection remains in its early stages, with model performance exhibiting variability across datasets. Drawing insights from other well-explored application areas like computer vision, we conjecture that the imbalance issue (the number of vulnerable code is extremely small) is at the core of the phenomenon. To validate this, we conduct a comprehensive empirical study involving nine open-source datasets and two state-of-the-art DL models. The results confirm our conjecture. We also obtain insightful findings on how existing imbalance solutions perform in vulnerability detection. It turns out that these solutions perform differently as well across datasets and evaluation metrics. Specifically: 1) *Focal loss* is more suitable to improve the precision, 2) *mean false error* and *class-balanced loss* encourages the recall, and 3) *random over-sampling* facilitates the F1-measure. However, none of them excels across all metrics. To delve deeper, we explore external influences on these solutions and offer insights for developing new solutions.

Keywords: Software security · Vulnerability detection · Deep learning · Imbalance.

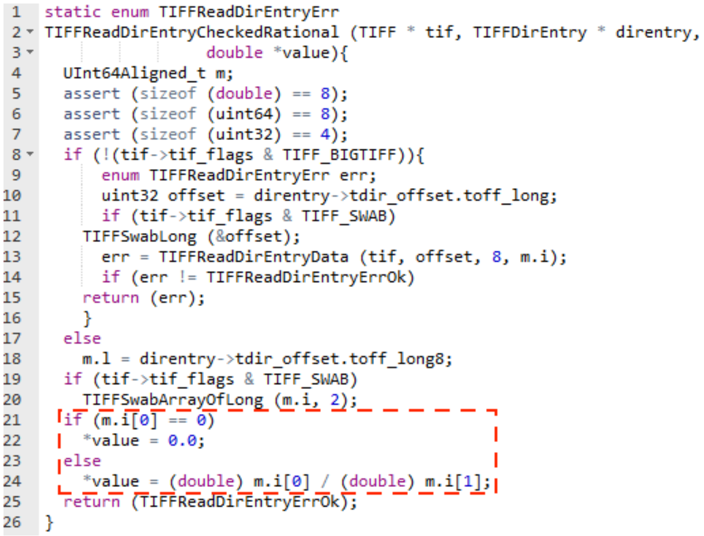
1 Introduction

The existence of software vulnerability is an inevitable risk in the software development life cycle, which raises significant concern since the vulnerability can be exploited by cybercriminals to run malicious code, install malware, and steal sensitive data. Discovering vulnerabilities in advance of the final deployment is ever required to enhance software security.

[*] This work is funded by the European Union's Horizon Research and Innovation Programme under Grant Agreement n°101070303.

[**] Corresponding author.

Manually identifying a function as vulnerable or not is tough concerning the required domain expertise and time. Fortunately, the rapid progress of deep learning (DL) largely automates this process [42]. In this paper, we are interested in applying DL to software vulnerability detection at the function level, which enables early detection of vulnerabilities during the programming stage. Figure 1 shows an example of a vulnerable function tagged with ID CVE-2017-7597[3]. The detailed information is that tif_dirread.c in LibTIFF 4.0.7 has an “outside the range of representable values of type float” undefined behavior issue, which might allow remote attackers to cause a denial of service (application crash) or possibly have unspecified other impacts via a crafted image.



```
static enum TIFFReadDirEntryErr
TIFFReadDirEntryCheckedRational (TIFF * tif, TIFFDirEntry * direntry,
                 double *value){
  UInt64Aligned_t m;
  assert (sizeof (double) == 8);
  assert (sizeof (uint64) == 8);
  assert (sizeof (uint32) == 4);
  if (!(tif->tif_flags & TIFF_BIGTIFF)){
      enum TIFFReadDirEntryErr err;
      uint32 offset = direntry->tdir_offset.toff_long;
      if (tif->tif_flags & TIFF_SWAB)
    TIFFSwabLong (&offset);
      err = TIFFReadDirEntryData (tif, offset, 8, m.i);
      if (err != TIFFReadDirEntryErrOk)
    return (err);
    }
  else
    m.l = direntry->tdir_offset.toff_long8;
  if (tif->tif_flags & TIFF_SWAB)
    TIFFSwabArrayOfLong (m.i, 2);
  if (m.i[0] == 0)
    *value = 0.0;
  else
    *value = (double) m.i[0] / (double) m.i[1];
  return (TIFFReadDirEntryErrOk);
}
```

Fig. 1: An example of vulnerable code: a C-language function from the LibTIFF project [41]. This function is tagged with the “Denial of Service” vulnerability in the Lin2018 dataset (please refer to Section 4.1 for more details). The code framed in a red rectangle highlights a concern about handling cases of division by zero when $m.i[1] == 0$.

Although various DL models [9,37,17,19] have been designed for vulnerability detection, the imbalance issue that causes a high false positive rate or false negative rate [10] is usually ignored. In this paper, “imbalance” refers to the numeral difference where the number of vulnerable code is much less than secure code in a dataset. In practice, software vulnerabilities do exist but rarely (e.g., 1 vulnerability in every 500 C-language functions) in source code programmed by experienced software developers. This imbalance problem is widely studied in other areas, such as computer vision (CV) and natural language processing

[3] https://www.cvedetails.com/cve/CVE-2017-7597/?q=CVE-2017-7597

(NLP), and to tackle it, both data and model-level methods have been proposed. The main idea is to put more importance on the minority set. Data-level methods straightforwardly down-sample [15] the majority set or over-sample [55] the minority set to ensure the numeral balance of data. Model-level approaches focus on the loss function [32,13] that guides the training procedure via adjusting the weights of majority and minority sets.

Although DL-based vulnerability detection is gaining attention, how the imbalance issue affects the "performance" of DL models in this specific area is still an open question. What makes things more complex is that, the model performance is evaluated by various metrics in the literature [26], such as accuracy [35], precision, and false positive rate. This makes it very difficult to compare their performance. For instance, the state-of-the-art (SOTA) model CodeBERT [35] is reported to achieve over 60% detection accuracy, while the false negative rate is 70%. The model is acceptable if accuracy is the evaluation criterion but useless in the case of a false negative rate.

In this paper, we conduct a comprehensive study on the imbalance issue in DL-based vulnerability detection. Through a series of experiments, our findings are summarized as follows. First of all, our first experimental results show that the imbalance problem tends to cause a model to gain a relatively low loss on secure code compared to the vulnerable one during the training procedure. Thus, The false negative rate is high.

Second, we experiment on how the imbalance solutions adapted from the CV and NLP domains perform in vulnerability detection. Our results show that:

– Compared to accuracy and false positive rate, precision, recall, and F1-measure are more informative when evaluating a DL model for vulnerability detection.

– None of the existing solutions from other domains performs perfectly across all selected datasets and models. Model-level solutions are beneficial to precision and recall. Data level solutions are more helpful to F1-measure.

Third, apart from the designed methodology, we explore how external factors affect the effectiveness of existing solutions, where these factors come from source code, such as the appearance of vulnerability types in the training procedure and test time and the detection difficulty of vulnerable code. Our experiment results show that external factors, such as the absence of vulnerabilities, identification difficulty of certain vulnerability types, and data distribution need to be considered when designing a new solution specifically for vulnerability detection.

For the readers to validate our findings, the experiment datasets and artifacts (including all solutions) for reproduction are made available on Git[4].

2 Background and Related Work

2.1 Software Vulnerability Detection

A software vulnerability is a security flaw and glitch found in source code. Detecting vulnerabilities has attracted considerable interest in the security community.

[4] `https://github.com/testing-cs/vulnerability-detection.git`

Manually checking source code is straightforward, but even for experts, this task is tedious and subjective because of great code complexity and diverse programming languages. At a higher level, there is the popular fuzzing [1] technique that automates the task. The basic idea of fuzzing is to generate a large number of test cases that are fed into the target program for execution. When a crash is triggered, it will be first determined as a bug or not and further identified as a vulnerability or not by exploitability analysis [16]. Since fuzzing highly relies on the generated test cases and the target program is required to be executed to monitor the behavior, it cannot be applied during a very early stage, such as the programming time.

Traditional detection tools like static analyzers [2] often require manual feature engineering by security researchers and target specific vulnerabilities, which is less efficient [43]. Deep learning (DL) facilitates the static analysis without running the target program and is feasible to function at different code module granularity [12], such as file level [18], function level [53,35,30,29], and program slice level [28,27]. Various DL models have been developed in the literature to support automated vulnerability detection. Simple examples [26] include multilayer perception (MLP), convolutional neural network (CNN), long short-term memory (LSTM), gated recurrent unit (GRU), bidirectional LSTM, and bidirectional GRU. Advanced ones dedicated to the structural representation of source code with graph neural networks, such as Devign [53] proposed by Zhou *et al.*. More recently, the application of foundation models is changing the domination of these task-specific models, which is discussed in the next subsection.

2.2 Recap of (foundation) DL Models

In classical DL, a model is initially randomly parameterized. Given a large set of labeled data, the model is trained with a certain number of epochs to achieve a satisfying performance for a given task. Therefore, this type of model is also called the task-specific model. Depending on the data type, model architecture, and learning task, the training procedure can take minutes or days. For instance, given the ImageNet-1k dataset to obtain an image classification model with ResNet-50, the 90-epoch training takes 14 days on a NVIDIA M40 GPU [51].

Different from task-specific models, foundation models, aka pre-trained models [20][5], break the limitation of relying on labeled data and have been a new paradigm of artificial intelligence (AI) [6]. Generally, a foundation model is trained using a huge volume of unlabeled data at scale and can be used for a wide range of downstream tasks. Via a few epochs' fine-tuning, the model can achieve SOTA performance. Foundation models have been increasingly developed and brought dramatic improvements in various communities, such as computer vision, natural language processing, and software engineering. Example models are the bidirectional encoder representations from transformers (BERT) [14],

[5] The term "foundation model" is used in this paper because, in the literature, a "pre-trained model" also has the meaning of a model trained by someone else and targeting a similar task[24,11].

generative pre-trained transformer 3 (GPT-3) [7], Roberta [33], ViLBERT [34], VideoBERT [45], CodeBERT [17], and GraphCodeBERT [19].

2.3 Solutions for Addressing Imbalance

There are mainly two types of methods for the imbalance issue, data level and model level [8]. Data level solutions focus on balancing the data size between minority and majority, such as down-sampling on majority and over-sampling on minority. The basic re-sampling strategy is in a random manner where samples are randomly selected to be removed [15] or duplicated [22]. An advanced over-sampling is to introduce new data based on neighboring samples [10] or synthesized [55]. However, advanced solutions [38] are data-dependent and some are inapplicable to source code. For instance, SMOTE [10] generates a new sample by joining k minority class nearest neighbors in the feature space, but generated code are likely to be invalid concerning both the syntax and semantics. Shu *et al.* [44] proposed Dazzle that leverages the Wasserstein Generative Adversarial networks as an over-sampling solution for software vulnerability detection. However, how to ensure the correctness of generate code is not addressed.

Model-level solutions, also known as cost-sensitive learning [13], assume the costs (represented by the loss) caused by different errors are unequal. The most common way is to put higher weight on the loss of the minority and less on the majority. Typical solutions include calculating the loss on minority and majority separately [49], effective number-based re-weighting [13], and misclassification-focused [32]. All these solutions adjust the decision boundary during the training procedure. Another method, threshold-moving [21] adjusts the boundary in the test time, which is simple but sensitive to the change in data.

Most existing solutions are initially proposed and studied in computer vision [13,32,8] and natural language processing [13,49]. In this paper, we investigate their effectiveness for software vulnerability detection.

3 Empirical Study Design

Figure 2 gives an overview of our study design. In total, four research questions are framed:
RQ1: How does imbalance impact model performance in vulnerability detection?
RQ2: Which metrics are appropriate for evaluating detection models?
RQ3: How effective are existing solutions in mitigating imbalance in vulnerability detection?
RQ4: What external factors may hinder the solutions to work?

3.1 Sketch of DL-based Vulnerability Detection Task

Generally, deep learning approaches formalize the vulnerability detection task as a binary classification problem, i.e., identifying a given code sample as secure (the label is 0) or vulnerable (the label is 1) [53]. Formally, let $\mathcal{D} = \{x, y\}$ be a

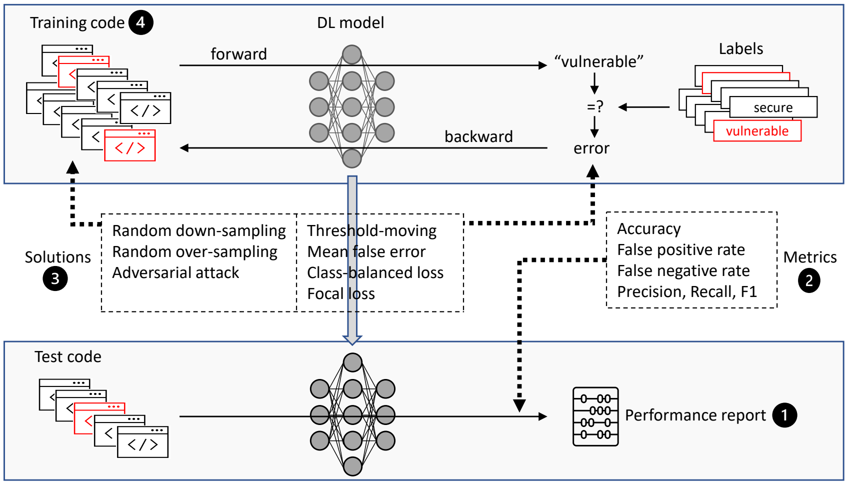


Fig. 2: Overview of the empirical study. During the training procedure (top), a foundation DL model is fine-tuned to minimize the error between predicted and ground truth labels. In the test time (bottom), the trained model is used to make predictions on test data. ❶❷❸❹ refer to four research questions.

source code set with samples, where $x \in X$ and $y \in Y$ represent a source code sample and its corresponding label ($Y = \{0, 1\}$), respectively. f_θ is such a binary classifier parameterized by θ that maps the data space X to the label space Y. The training procedure of the model is to find the optimal θ^* that minimizes the error between prediction and ground truth as shown in Figure 2. Formally, the optimization objective is defined by:

$$\theta^* = \underset{\theta \in \Theta}{\arg\min}\, \ell \left(f_\theta, X, Y\right) \tag{1}$$

where ℓ is a loss function that captures the error between the predicted labels by f_θ and ground truth labels. The loss function can be in different forms, such as cross-entropy loss, mean squared error, and hinge embedding loss. For vulnerability detection, the cross-entropy loss is mostly applied and in this form, the objective becomes to minimize the cross entropy over all samples:

$$\ell = \frac{1}{N} \sum_{i=1}^{N} CE\left(p_i, y_i\right) \tag{2}$$

where

$$CE\left(p_i, y_i\right) = \begin{cases} -\log\left(p_i\right) & y_i = 1 \\ -\log\left(1 - p_i\right) & y_i = 0 \end{cases} \tag{3}$$

Note that in Eq. (2), the loss is an average over all samples. That is, all samples are equally treated regardless of being secure or vulnerable, which is reasonable when data are evenly distributed in two types. However, in reality, the number of vulnerable code is usually much less than secure ones, which causes the imbalance issue for training.

Remarkably, let p $(0 \leq p \leq 1)$ be the predicted probability of x being vulnerable. In binary classifiers, usually, a decision threshold is set at 0.5. Namely, if $p > 0.5$, x is determined as vulnerable, otherwise secure.

3.2 Imbalance Solutions

In total, we examine seven methods that are widely adopted in the literature to handle the imbalance problem. These methods cover both the data and model-level solutions. Note that data level (marked with *) solutions only carry out on the training set during the training time. Let N_s and N_v denote the number of secure and vulnerable code, respectively. $N_d = N_s - N_v$.

***Random down-sampling** [15]. N_d secure code are randomly selected and removed from the training set.

***Random over-sampling** [22]. As opposed to down-sampling, N_d vulnerable code are randomly selected from and replicated to the training set.

***Adversarial attack-based augmentation** [50,52]. An advanced over-sampling method that generates new vulnerable code via adversarial attack. In this paper, we perform the variable renaming-based adversarial attack that changes the name of local variables. The advantage is that the generated code can pass the compiler and remain executable. To ensure that the substitute name is natural to software developers, we use the masked language prediction function of CodeBERT [17] to produce the candidates.

Threshold-moving [21], also known as threshold-tuning, post-scaling, and thresholding. It adjusts the decision threshold and is applied during the test time. Concretely, the model is trained using the training set and then used to predict the probabilities of samples in the validation set. Given a candidate threshold set, the performance on the validation set is evaluated on each candidate and the threshold producing the best performance is selected as the optimal one for the prediction on the test set. In this paper, we set the candidate threshold to range from 0 to 1 with a 0.01 interval.

Mean false error (MFE) loss [49]. The concept comes from the "false positive rate" and "false negative rate". The goal is to make the loss more sensitive to the errors caused by the minority set by computing the loss on different sets separately. The original base loss is the mean squared error for image classification and document classification tasks. We adapt it to the cross entropy error to fit vulnerability detection models. Formally,

$$\ell_{mfe} = \frac{1}{N_s} \sum_{i=1}^{N_s} CE\left(p_i, y_i\right) + \frac{1}{N_v} \sum_{j=1}^{N_v} CE\left(p_j, y_j\right) \tag{4}$$

Class-balanced (CB) loss [13]. This method addresses the imbalance issue by introducing the effect number that refers to the expected volume of samples of a given set. Formally,

$$\ell_{cb} = \frac{1}{N} \sum_{i=1}^{N} \frac{1-\beta}{1-\beta^{N_{y_i}}} CE\left(p_i, y_i\right) \tag{5}$$

where $N_{y_i} = N_v$ if $y_i = 1$ otherwise N_s. As recommended [13], we set $\beta = 0.9999$.

Focal loss (FL) [32]. The idea is to put more focus on hard, misclassified samples meanwhile reduce the loss for well-classified samples. For instance, given three vulnerable code x_1, x_2, and x_3, the model predicts them as vulnerable with 0.9, 0.5, and 0.2 probability, respectively. x_1 is well-classified. x_2 is hard to classify. x_3 is misclassified as secure. Formally,

$$\ell_{fl} = -\frac{1}{N} \sum_{i=1}^{N} \left(1-p_t\right)^{\gamma} \log\left(p_t\right) \tag{6}$$

where

$$p_t = \begin{cases} p_i & y_i = 1 \\ 1-p_i & y_i = 0 \end{cases} \tag{7}$$

when $\gamma = 0$, Eq. (6) is equivalent to Eq. (2). As recommended [32], we set $\gamma = 2$.

3.3 Evaluation Metrics

We investigate all the following six metrics which have been used in different papers to evaluate software vulnerability detection models [9,17,19,53,28].

Accuracy [17,19,53] is the percentage of samples that are correctly classified by a model. This metric may give a fake good performance. For instance, if a test set has 100 code where only one is vulnerable. A model classifies all samples as secure, so its accuracy is 99%, which is nearly perfect. However, this model is useless as a detection model.

False positive rate (FPR) [28] measures the ratio of misclassified secure code to the total number of secure samples. A low value means the model learns very well from secure code.

False negative rate (FNR) [28] computes the ratio of misclassified vulnerable code to the total number of vulnerable samples. This metric focuses on the ability to figure out vulnerable code. A low value indicates a strong ability.

Precision [53], also known as positive predictive rate, is the fraction of correctly classified vulnerable code among samples classified as vulnerable.

Recall [53], the opposite of FNR, is the fraction of correctly classified vulnerable code among all vulnerable samples. In practice, precision and recall are often in tension. Improving precision will cause recall to decay, and vice versa.

F1-measure (F1) [53] is defined as the harmonic mean of precision and recall. It balances the importance between precision and recall.

4 Experimental Setup

All experiments were conducted on a high-performance computer (HPC) cluster and each cluster node runs a 2.20GHZ Intel Xeon Silver 4210 GPU with an NVIDIA Tesla V100 32G GPU. Models are trained and tested using the PyTorch 1.7.1 framework with CUDA 10.1.

4.1 Datasets

As listed in Table 1, nine function-level datasets from three open-source repositories on GitHub are considered in the experiments. All the datasets are C-language and the related projects are popular among software developers. Devign [40] is provided by Zhou *et al.* [54] and consists of two datasets collected from the FFmpeg [5] and QEMU [4] projects, respectively. Labels of source code are manually annotated by professional security researchers. Lin2018 [31][6] includes six datasets from Asterisk [3], FFmpeg [5], LibPNG [46], LibTIFF [41], Pidgin [39], and VLC media player [48], respectively. For each project, source code are manually labeled by Lin *et al.* according to the CVE and NVD records. CodeXGLUE [36] provides a mixture version of two datasets from FFmpeg and QEMU in Devign.

Table 1: Datasets overview. IR: imbalance ratio ($\frac{\#Secure}{\#Vulnerable}$).

Source	**Project**	**Project description**	**#Vulnerable**	**#Secure**	**#Total**	**IR**
Devign	FFmpeg	A cross-platform to record, convert and stream audio and video.	4,981	4,788	9,769	0.96
	QEMU	A generic and open source machine emulator and virtualizer.	7,479	10,070	17,549	1.35
Lin2018	Asterisk	A framework for building communications applications.	56	17,070	17,126	304.82
	FFmpeg	A cross-platform to record, convert and stream audio and video.	213	5,550	5,763	26.06
	LibPNG	Official PNG reference.	44	577	621	13.11
	LibTIFF	TIFF library and utilities.	96	731	827	7.61
	Pidgin	A multi-platform instant messaging client.	29	8,612	8,641	296.97
	VLC	A cross-platform multimedia player.	43	6,113	6,156	142.16
CodeXGLUE	Devign	Mixture of FFmpeg and QEMU.	12,460	14,858	27,318	1.19

To have a closer view of the vulnerabilities existing in these datasets, we provide Table 2. In total, 25 vulnerabilities are included and a certain vulnerability may have more than one CVE record with different CVSS scores, e.g., Bypass a restriction or similar.

4.2 Models

Two SOTA foundation models, CodeBERT [17] and GraphCodeBERT [19], for natural language and programming language, are leveraged in this paper. Both models follow BERT [14] and use multi-layer bidirectional Transformer [47] as

[6] Notice: the number of data is a bit different from the original paper in [30] because we remove empty source code files from the provided datasets. Empty files cause compiling bugs and degrade the model performance.

Table 2: List of vulnerabilities. The common vulnerability scoring system (CVSS) score measures the severity of a certain vulnerability type.

ID	Vulnerability Type	CVSS	ID	Vulnerability Type	CVSS
1	Bypass a restriction or similar	4.3 - 7.5	14	Execute Code	5.8 - 9.3
2	Cross Site Scripting	4.3	15	Execute Code Gain privileges	6.5
3	Denial Of Service	2.6 - 7.8	16	Execute Code Memory corruption	6.8
4	Denial Of Service Execute Code	6.8 - 9.3	17	Execute Code Memory corruption Obtain Information	6.8
5	Denial Of Service Execute Code Memory corruption	6.8 - 10.0	18	Execute Code Overflow	6.0 - 10.0
6	Denial Of Service Execute Code Overflow	6.5 - 10.0	19	Execute Code Overflow Bypass a restriction or similar	6.8
7	Denial Of Service Execute Code Overflow Memory corruption	6.8	20	Execute Code Overflow Memory corruption	6.8
8	Denial Of Service Memory corruption	6.8	21	Gain privileges	9.0
9	Denial Of Service Obtain Information	4.3 - 10	22	Obtain Information	4.3 - 5.0
10	Denial Of Service Overflow	2.6 - 9.3	23	Overflow	4.3 - 10
11	Denial Of Service Overflow Memory corruption	4.3 - 6.8	24	Overflow Memory corruption	5.0
12	Denial Of Service Overflow Obtain Information	5.8	25	Unspecified	4.3 - 10.0
13	Directory traversal	5.8 - 9.3			

the backbone. CodeBERT is pre-trained on 2.1M bimodal data and 6.4M unimodal codes. GraphCodeBERT is pre-trained on the CodeSearchNet [23] dataset consisting of 2.3M functions paired with natural language descriptions. The main difference between CodeBERT and GraphCodeBERT is that the source code in CodeBERT is represented as a sequence of tokens, while GraphCodeBERT takes the data flow of source code as its input.

Our implementation is adapted from the GitHub repositories provided by CodeXGLUE[7] for CodeBERT and by Microsoft[8] for GraphCodeBERT, respectively. The base models for fine-tuning are loaded from Hugging Face[9][10] from Hugging Face.

4.3 Training

Each model is fine-tuned 50 epochs and the “best” one is saved for evaluation. For reproduction, we follow the default setting in original implementations to set the random seed at 123456. In each dataset, we proportionally (8:1:1) split the dataset into a training set, a validation set, and a test set (the training and validation sets are involved in the training procedure, and the test set is only for testing.). Vulnerable and secure code are randomly divided into these three sets with the same imbalance ratio as in Table 1.

5 Results

5.1 RQ1: Influence of Imbalance in Vulnerability Detection

Experiments. We train CodeBERT and GraphCodeBERT with default settings, ignoring the imbalance. For each trained model, we check if the imbalance causes a bias towards secure code by comparing loss and accuracy across individual sets.

[7] `https://github.com/microsoft/CodeXGLUE`

[8] `https://github.com/microsoft/CodeBERT`

[9] `https://huggingface.co/microsoft/codebert-base`

[10] `https://huggingface.co/microsoft/graphcodebert-base`

Table 3: Model accuracy and loss on secure and vulnerable code. **Baseline**: model accuracy on all code. The best performance is highlighted.

Source	**Project**	**Accuracy (%)**			**Total Loss**		**Average loss**	
		Baseline	**Vulnerable**	**Secure**	**Vulnerable**	**Secure**	**Vulnerable**	**Secure**
				CodeBERT				
Devign	FFmpeg	56.71	62.17	51.04	537.37	449.77	0.72	0.63
	QEMU	64.31	40.96	81.67	1361.10	583.43	1.21	0.39
Lin2018	Asterisk	99.77	44.44	99.96	33.75	3.18	3.75	0.00
	FFmpeg	97.46	36.36	99.88	124.56	2.92	3.77	0.00
	LibPNG	95.83	50.00	100.00	14.25	0.65	1.78	0.01
	LibTIFF	88.89	53.33	93.69	49.34	30.54	3.29	0.28
	Pidgin	99.85	60.00	100.00	17.14	0.08	3.43	0.00
	VLC	99.89	85.71	100.00	3.07	0.08	0.44	0.00
CodeXGLUE	Devign	61.49	31.95	86.59	1716.89	531.24	1.37	0.36
				GraphCodeBERT				
Devign	FFmpeg	56.99	74.87	38.39	437.05	559.02	0.58	0.78
	QEMU	65.38	52.98	74.59	1001.18	776.03	0.89	0.51
Lin2018	Asterisk	99.81	44.44	100.00	31.18	0.90	3.46	0.00
	FFmpeg	97.35	51.52	99.16	134.98	37.00	4.09	0.04
	LibPNG	96.88	62.50	100.00	16.16	0.68	2.02	0.01
	LibTIFF	92.06	33.33	100.00	47.81	1.31	3.19	0.01
	Pidgin	99.85	60.00	100.00	13.18	0.39	2.64	0.00
	VLC	99.68	57.14	100.00	11.77	1.26	1.68	0.00
CodeXGLUE	Devign	62.81	58.25	66.69	1047.34	806.85	0.83	0.55

Results. Table 3 shows the results. Regardless of the model and evaluation metric, the model performs better on the secure set than on the vulnerable one. Considering the accuracy, except for the FFmpeg dataset in Devign, both models achieve higher accuracy on secure code. Particularly, 100% secure code can be perfectly identified in several datasets, such as LibPNG, LibTIFF, Pidgin, and VLC. However, the performance in identifying vulnerable code is less satisfying. For instance, in Asterisk from Lin2018, GraphCodeBERT can correctly identify all secure code but only 44.44% vulnerable code. On the other hand, with respect to the loss between prediction and ground truth, the loss on secure code is, in general, much lower than on vulnerable code. For instance, on average, CodeBERT has no loss on each secure code for Pidgin from Lin2018, but 0.44 on the vulnerable one. This indicates that during the training procedure, the model tends to learn more from the secure code. When summing over all code, the imbalance makes it worse, e.g., GraphCodeBERT has, in total, 0.39 loss on secure code but 13.18 on vulnerable one. The reason is that, during the training procedure, the loss is calculated as an average (Eq. (2)) or sum over all samples (both the majority secure and minority vulnerable). This methodology weakens the influence of vulnerable code and gives “fake” feedback to the training that the model is performing well, which is the essence of the imbalance issue.

Answer: The imbalance encourages a DL model to gain more knowledge from secure code, which leads to poor performance on detecting vulnerable code, e.g., 44.44% accuracy (55.56% false negative rate).

5.2 RQ2: Analysis of Evaluation Metrics

Experiments. In the default setting of CodeBERT and GraphCodeBERT, for each trained model, different metrics are used to evaluate the model performance.

Results. Table 4 lists the model performance. From the perspective of identifying vulnerable code, accuracy and FPR are not informative enough and can be misleading. For instance, Table 3 shows that CodeBERT only successfully detects 44.44% vulnerable code in Asterisk from Lin2018. However, the output accuracy is almost perfect at 99.77%. The overall accuracy hides the actual performance of detecting vulnerable code. FPR only considers the detection of secure code, which misses the main purpose of vulnerable detection, namely to identify vulnerabilities at an early stage. Recall (the opposite of FNR) is equivalent to the individual accuracy on the vulnerable code in Table 3 and can tell how the model identifies vulnerable code. However, recall ignores the secure code. Precision covers this shortage by including misclassified secure code. If one only cares about detecting vulnerable code and ignores the cost of manually filtering secure code afterward, recall is the best option. If one wishes to have fewer errors in the identified vulnerable code, precision can be taken. As a balanced version between precision and recall, F1 can be used when an overall score is preferred.

Table 4: Model performance (%) using different evaluation metrics. Accuracy is the default metric of CodeBERT and GraphCodeBERT.

Source	**Project**	**Accuracy**	**FPR**	**FNR**	**Precision**	**Recall**	**F1**
			CodeBERT				
Devign	FFmpeg	56.71	48.96	37.83	56.92	62.17	59.42
	QEMU	64.31	18.33	59.04	62.42	40.96	49.46
Lin2018	Asterisk	99.77	0.04	55.56	80.00	44.44	57.14
	FFmpeg	97.46	0.12	63.64	92.31	36.36	52.17
	LibPNG	95.83	0.00	50.00	100.00	50.00	66.67
	LibTIFF	88.89	6.31	46.67	53.33	53.33	53.33
	Pidgin	99.85	0.00	40.00	100.00	60.00	75.00
	VLC	99.89	0.00	14.29	100.00	85.71	92.31
CodeXGLUE	Devign	61.49	13.41	68.05	66.94	31.95	43.26
			GraphCodeBERT				
Devign	FFmpeg	56.99	61.61	25.13	55.83	74.87	63.96
	QEMU	65.38	25.41	47.02	60.78	52.98	56.61
Lin2018	Asterisk	99.81	0.00	55.56	100.00	44.44	61.54
	FFmpeg	97.35	0.84	48.48	70.83	51.52	59.65
	LibPNG	96.88	0.00	37.50	100.00	62.50	76.92
	LibTIFF	92.06	0.00	66.67	100.00	33.33	50.00
	Pidgin	99.85	0.00	40.00	100.00	60.00	75.00
	VLC	99.68	0.00	42.86	100.00	57.14	72.73
CodeXGLUE	Devign	62.81	33.31	41.75	59.77	58.25	59.00

Answer: Precision, recall and F1 provide more informative and comprehensive insights on model performance than accuracy. FPR might be useful in some situations to limit the impact of false positives (e.g., static analysis), but precision can serve a similar purpose as a higher precision generally implies a lower FPR.

5.3 RQ3: Effectiveness of Solutions for Addressing Imbalance

Experiments. We train CodeBERT and GraphCodeBERT following the methodology of different solutions for handling the imbalance issue. Based on the selected evaluation metrics, precision, recall, and F1, by RQ2, the effectiveness of solutions is investigated.

Results. Table 5 and Table 6 show the results on CodeBERT and GraphCodeBERT, respectively. Note that in FFmpeg, Devign, the number of vulnerable programs (4981) is greater than the secure one (4788), thus, no re-sampling-based solutions are applied. Regardless of the dataset, model, and evaluation metric, random down-sampling performs the worst since massive information about the secure code is eliminated. In particular, when the imbalance ratio is high and the data size is small (e.g., Asterisk from Lin2018), the remaining data is insufficient to support the model training. The focal loss stands out as the optimal choice for improving precision. The reason is that focal loss puts more effort into hard and misclassified samples during the training procedure whether those samples are vulnerable or secure. Thus, the model can more precisely predict a code sample to be vulnerable or secure. Two model-level solutions, MFE and CB, are the worst regarding precision. The reason is that, the methodology of these two solutions is to put relatively more attention to the vulnerable code during the training procedure, thus, more vulnerable code should be correctly identified than the baseline. This is confirmed by the results of recall where both solutions outperform the others. While in this case, the focal loss gains low recall. With respect to the overall performance F1, random over-sampling seems to be the best in most cases for both models.

Answer: No single existing solution is the best to address the imbalance issue across all evaluation metrics. Specifically, to focal loss is the best option for improving precision. MFE and CB shall be used for optimizing recall. Random over-sampling is the best option when focusing on the overall F1 performance. Nevertheless, the pursuit of a new, task-specific solution to address the imbalance issue remains imperative.

5.4 RQ4: Investigation of External Factors

Experiments. Based on Table 5 and Table 6, we dig into the prediction results to explore possible external factors.

Results. Note that in some cases, the model performance degrades after applying a solution. For instance, in Pidgin from Lin2018, CodeBERT with the default setting gains 100% precision and 60% recall, but 75% precision and the same recall by adversarial attack-based augmentation. We found that by over-R, over-A, thresholding, and MFE, CodeBERT identifies the same vulnerable samples as the baseline, and the code with the Overflow vulnerability (ID 23

Table 5: CodeBERT trained using different solutions for imbalance issues. **Baseline**: the default setting. **Down-R**: random down-sampling. **Over-R**: random over-sampling. **Over-A**: adversarial attack-based augmentation. For each dataset, the best solution under a given metric is highlighted.

Source	**Project**	**Baseline**	**Down-R**	**Over-R**	**Over-A**	**Thresholding**	**MFE**	**CB**	**FL**
					Precision				
Devign	FFmpeg	56.92	-	-	-	0.00	67.72	63.14	88.24
	QEMU	62.42	60.51	62.33	71.12	0.00	55.36	55.17	93.04
Lin2018	Asterisk	80.00	0.00	75.00	100.00	0.00	66.67	54.55	100.00
	FFmpeg	92.31	3.23	76.19	65.38	100.00	34.29	38.71	90.00
	LibPNG	100.00	66.67	100.00	85.71	0.00	62.50	70.00	100.00
	LibTIFF	53.33	66.67	60.00	80.00	72.73	57.14	53.33	80.00
	Pidgin	100.00	0.00	100.00	75.00	100.00	60.00	0.00	0.00
	VLC	100.00	0.00	100.00	100.00	100.00	0.00	0.00	0.00
CodeXGLUE	Devign	66.94	59.52	62.03	62.72	100.00	58.38	59.82	85.49
					Recall				
Devign	FFmpeg	62.17	-	-	-	0.00	31.42	47.86	4.01
	QEMU	40.96	50.49	49.07	29.39	0.00	71.33	75.96	9.53
Lin2018	Asterisk	44.44	0.00	66.67	44.44	0.00	44.44	66.67	33.33
	FFmpeg	36.36	3.03	48.48	51.52	36.36	72.73	72.73	54.55
	LibPNG	50.00	25.00	62.50	75.00	0.00	62.50	87.50	62.50
	LibTIFF	53.33	40.00	40.00	26.67	53.33	53.33	53.33	26.67
	Pidgin	60.00	0.00	60.00	60.00	60.00	60.00	0.00	0.00
	VLC	85.71	0.00	85.71	57.14	85.71	0.00	0.00	0.00
CodeXGLUE	Devign	31.95	47.33	52.59	45.98	5.02	53.55	47.81	13.15
					F1				
Devign	FFmpeg	59.42	-	-	-	0.00	42.92	54.45	7.67
	QEMU	49.46	55.05	54.91	41.59	0.00	62.33	63.92	17.29
Lin2018	Asterisk	57.14	0.00	70.59	61.54	0.00	53.33	60.00	50.00
	FFmpeg	52.17	3.13	59.26	57.63	53.33	46.60	50.53	67.92
	LibPNG	66.67	36.36	76.92	80.00	0.00	62.50	77.78	76.92
	LibTIFF	53.33	50.00	48.00	40.00	61.54	55.17	53.33	40.00
	Pidgin	75.00	0.00	75.00	66.67	75.00	60.00	0.00	0.00
	VLC	92.31	0.00	92.31	72.73	92.31	0.00	0.00	0.00
CodeXGLUE	Devign	43.26	52.73	56.92	53.06	9.56	55.86	53.14	22.79

in Table 2) is always misclassified. This is because this vulnerability type does not appear in the training or validation sets (as shown in Figure 3(b)) and the model cannot gain knowledge of this specific vulnerability type. Introducing more vulnerable samples can just cause the overfitting problem. Another case is in LibTIFF from Lin2018, the thresholding, MFE, and CB identify the same vulnerable samples as the baseline and miss five vulnerability types, Denial Of Service (ID 3,) Denial Of Service Execute Code Overflow (ID 6), Denial Of Service Overflow (ID 10), Execute Code Overflow (ID 18), and Overflow (ID 23). All these types are included in the training procedure (training and validation sets) (see Figure 3(a)). All the trained models with or without solutions reach 53.33% recall, these solutions can only increase the correctness of secure code because, by the corresponding training methodology, the model already reaches the limit of identifying certain types of vulnerability. In addition, thresholding

Table 6: GraphCodeBERT trained using different solutions for imbalance issue. **Baseline**: the default setting. **Down-R**: random down-sampling. **Over-R**: random over-sampling. **Over-A**: adversarial attack-based augmentation. For each dataset, the best solution under a given metric is highlighted.

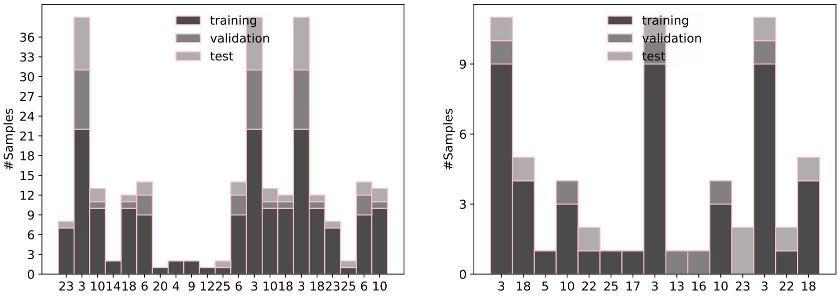
Source	Project	Baseline	Down-R	Over-R	Over-A	Thresholding	MFE	CB	FL
				Precision					
Devign	FFmpeg	55.83	-	-	-	0.00	59.48	56.76	86.36
	QEMU	60.78	58.43	61.19	65.28	93.33	56.70	54.93	85.15
Lin2018	Asterisk	100.00	0.00	44.44	100.00	100.00	38.46	23.53	100.00
	FFmpeg	70.83	24.53	70.97	70.37	75.00	28.00	17.39	62.50
	LibPNG	100.00	66.67	83.33	85.71	0.00	66.67	66.67	100.00
	LibTIFF	100.00	0.00	100.00	77.78	0.00	55.56	47.06	100.00
	Pidgin	100.00	0.00	100.00	75.00	0.00	50.00	20.00	66.67
	VLC	100.00	0.00	83.33	100.00	0.00	75.00	75.00	100.00
CodeXGLUE	Devign	59.77	58.25	59.75	60.05	100.00	60.36	60.86	90.21
				Recall					
Devign	FFmpeg	74.87	-	-	-	0.00	52.41	65.11	5.08
	QEMU	52.98	54.94	53.07	37.67	6.23	64.02	62.51	15.32
Lin2018	Asterisk	44.44	0.00	44.44	44.44	33.33	55.56	44.44	22.22
	FFmpeg	51.52	78.79	66.67	57.58	45.45	84.85	84.85	60.61
	LibPNG	62.50	75.00	62.50	75.00	0.00	75.00	75.00	62.50
	LibTIFF	33.33	0.00	40.00	46.67	0.00	66.67	53.33	26.67
	Pidgin	60.00	0.00	80.00	60.00	0.00	80.00	60.00	80.00
	VLC	57.14	0.00	71.43	85.71	0.00	85.71	85.71	71.43
CodeXGLUE	Devign	58.25	55.70	56.89	54.50	4.54	61.04	57.85	13.94
				F1					
Devign	FFmpeg	63.96	-	-	-	0.00	55.72	60.65	9.60
	QEMU	56.61	56.63	56.84	47.77	11.69	60.14	58.48	25.96
Lin2018	Asterisk	61.54	0.00	44.44	61.54	50.00	45.45	30.77	36.36
	FFmpeg	59.65	37.41	68.75	63.33	56.60	42.11	28.87	61.54
	LibPNG	76.92	70.59	71.43	80.00	0.00	70.59	70.59	76.92
	LibTIFF	50.00	0.00	57.14	58.33	0.00	60.61	50.00	42.11
	Pidgin	75.00	0.00	88.89	66.67	0.00	61.54	30.00	72.73
	VLC	72.73	0.00	76.92	92.31	0.00	80.00	80.00	83.33
CodeXGLUE	Devign	59.00	56.95	58.29	57.14	8.69	60.70	59.31	24.15

tends to ruin the model entirely, such as FFmpeg from Devign and LibPNG from Lin2018, which is caused by the distribution shift between the validation set in the training time and the test set in the test time. Distribution shift [25] is a research topic per se and is not further explained in this paper.

Answer: External factors including the absence of vulnerability types in the training time, inherent identification difficulty of certain vulnerability types, and the distribution shift in data should be considered when developing a new solution.

5.5 Insights

Selecting evaluation metrics: In vulnerability detection, when selecting a metric to evaluate a model's performance, accuracy is the least suitable metric. Recall should be used if only the detection on vulnerable code matters. Precision



(a) LibTIFF

(b) Pidgin

Fig. 3: Vulnerability type distribution in each split set (training, validation, and test). x-axis: vulnerability type ID (Please refer to Table 2 for more details.). y-axis: number of samples in the corresponding set. Source: Lin2018.

should be selected if one wishes to have less secure code in identified vulnerable code. F1 can be considered from an overall perspective.

Designing solutions: When designing a solution to address imbalance, one should consider the evaluation metric first. If the goal is to improve precision or recall, modifying the loss function is more efficient than manipulating training data. Minority over-sampling brings benefits to the overall evaluation. Vulnerability type and difficulty in data should be considered when a solution fails.

6 Conclusion

This work studies the imbalance issue in software vulnerability detection. Seven solutions proposed in other domains are investigated on nine open-source datasets and two state-of-the-art deep learning models (CodeBERT and GraphCodeBERT). We found the defaulting setting of CodeBERT and GraphCodeBERT makes the training procedure focus more on the secure code, which causes a high false negative rate (e.g., 68.05%). Existing solutions perform differently over various datasets and models, which calls for a new solution specifically for vulnerability detection. With the insights stated in the paper, this will be an interesting future work. Furthermore, we explore external factors like the vulnerability type distribution that should be aware of when designing such a new solution. There are many future research topics. The observations from this paper should be tested on other datasets and for other programming languages. Related to this, the observations should also be tested on other ML models other than CodeBERT and GraphCodeBERT. External factors, which can affect the performances, should be explored in more depth. This is particularly important if a solution is about to be deployed in practice.

References

1. Amankwah, R., Kudjo, P., Yeboah, S.: Evaluation of software vulnerability detection methods and tools: a review. International Journal of Computer Applications **169**, 22–27 (July 2017). https://doi.org/10.5120/ijca2017914750
2. Arusoaie, A., Ciobâca, S., Craciun, V., Gavrilut, D., Lucanu, D.: A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. In: 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. pp. 161–168. IEEE (2017). https://doi.org/10.1109/SYNASC.2017.00035
3. Asterisk team: Asterisk website. https://www.asterisk.org/ (2022), online, accessed on 25 August 2023
4. Bellard, F.: Qemu wesite. https://www.qemu.org/ (2022), online, accessed on 25 August 2023
5. Bellard, F., FFmpeg team: Repository of ffmpeg on github. https://github.com/FFmpeg/FFmpeg (2023), online, accessed on 25 August 2023
6. Bommasani, R., Hudson, D.A., Adeli, E., et al.: On the opportunities and risks of foundation models. CoRR **abs/2108.07258** (2021), https://arxiv.org/abs/2108.07258
7. Brown, T., Mann, B., Ryder, N., et al.: Language models are few-shot learners. In: Advances in Neural Information Processing Systems. pp. 1877–1901. Curran Associates, Inc. (2020), https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf
8. Buda, M., Maki, A., Mazurowski, M.A.: A systematic study of the class imbalance problem in convolutional neural networks. Neural Networks **106**, 249–259 (2018). https://doi.org/https://doi.org/10.1016/j.neunet.2018.07.011
9. Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: are we there yet? IEEE Transactions on Software Engineering **48**(09), 3280–3296 (September 2022). https://doi.org/10.1109/TSE.2021.3087402
10. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority over-sampling technique. Journal of Artificial Intelligence Research **16**(1), 321–357 (June 2002). https://doi.org/10.1613/jair.953
11. Choi, S., Yang, S., Choi, S., Yun, S.: Improving test-time adaptation via shift-agnostic weight regularization and nearest source prototypes. In: Computer Vision – ECCV 2022. pp. 440–458. Springer Nature Switzerland, Cham (2022). https://doi.org/10.1007/978-3-031-19827-4_26
12. Croft, R., Xie, Y., Babar, M.A.: Data preparation for software vulnerability prediction: a systematic literature review. IEEE Transactions on Software Engineering **49**, 1044–1063 (March). https://doi.org/10.1109/TSE.2022.3171202
13. Cui, Y., Jia, M., Lin, T.Y., Song, Y., Belongie, S.: Class-balanced loss based on effective number of samples. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 9260–9269. IEEE (2019). https://doi.org/10.1109/CVPR.2019.00949
14. Devlin, J., Chang, M., Lee, K., Toutanova, K.: Bert: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. pp. 4171–4186. Association for Computational Linguistics (2019), https://aclanthology.org/N19-1423.pdf
15. Drummond, C., Holte, R.: C4.5, class imbalance, and cost sensitivity: why under-sampling beats oversampling. In: International Conference on Machine Learning Workshop on Learning from Imbalanced Data Sets II. Washington, DC, USA (July 2003), https://www.site.uottawa.ca/~nat/Workshop2003/drummondc.pdf

16. Fell, J.: A review of fuzzing tools and methods. PenTest Magazine, March (2017)
17. Feng, Z., Guo, D., Tang, D., et al.: Codebert: a pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1536–1547. Association for Computational Linguistics (November 2020). https://doi.org/10.18653/v1/2020.findings-emnlp.139
18. Garg, A., Degiovanni, R., Jimenez, M., Cordy, M., Papadakis, M., Le Traon, Y.: Learning from what we know: How to perform vulnerability prediction using noisy historical data. Empirical Software Engineering **27**(7) (December 2022). https://doi.org/10.1007/s10664-022-10197-4
19. Guo, D., Ren, S., Lu, S., et al.: Graphcodebert: pre-training code representations with data flow. In: International Conference on Learning Representations (2021), https://openreview.net/pdf?id=jLoC4ez43PZ
20. Han, X., Zhang, Z., Ding, N., et al.: Pre-trained models: past, present and future. AI Open **2**, 225–250 (2021). https://doi.org/10.1016/j.aiopen.2021.08.002
21. He, H., Ma, Y.: Imbalanced learning: foundations, algorithms, and applications. Wiley-IEEE Press, 1st edn. (2013)
22. Huang, C.Y., Dai, H.L.: Learning from class-imbalanced data: review of data driven methods and algorithm driven methods. Data Science in Finance and Economics **1**(1), 21–36 (2021). https://doi.org/10.3934/DSFE.2021002
23. Husain, H., Wu, H.H., Gazit, T., Allamanis, M., Brockschmidt, M.: Codesearchnet challenge: evaluating the state of semantic code search. CoRR **abs/1909.09436** (2019), http://arxiv.org/abs/1909.09436
24. Kim, J., Feldt, R., Yoo, S.: Guiding deep learning system testing using surprise adequacy. In: 41st International Conference on Software Engineering. p. 1039–1049. IEEE Press (2019). https://doi.org/10.1109/ICSE.2019.00108
25. Koh, P.W., Sagawa, S., Marklund, H., et al.: Wilds: a benchmark of in-the-wild distribution shifts. In: 38th International Conference on Machine Learning. pp. 5637–5664. PMLR (July 2021)
26. Li, Z., Zou, D., Tang, J., Zhang, Z., Sun, M., Jin, H.: A comparative study of deep learning-based vulnerability detection system. IEEE Access **7**, 103184–103197 (2019). https://doi.org/10.1109/ACCESS.2019.2930578
27. Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z.: Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing **19**(04), 2244–2258 (July 2022). https://doi.org/10.1109/TDSC.2021.3051525
28. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: a deep learning-based system for vulnerability detection. In: 25th Annual Network and Distributed System Security Symposium. The Internet Society (February 2018), http://dx.doi.org/10.14722/ndss.2018.23158
29. Lin, G., Xiao, W., Zhang, J., Xiang, Y.: Deep learning-based vulnerable function detection: a benchmark. In: Information and Communications Security. pp. 219–232. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-41579-2_13
30. Lin, G., Zhang, J., Luo, W., Pan, L., Xiang, Y., De Vel, O., Montague, P.: Cross-project transfer representation learning for vulnerable function discovery. IEEE Transactions on Industrial Informatics **14**(7), 3289–3297 (2018). https://doi.org/10.1109/TII.2018.2821768
31. Lin, G., Zhang, J., Luo, W., Pan, L., Xiang, Y., De Vel, O., Montague, P.: Repository of lin2018 on github. https://github.com/DanielLin1986/TransferRepresentationLearning (2019), online, accessed on 25 August 2023

32. Lin, T.Y., Goyal, P., Girshick, R., He, K., Dollár, P.: Focal loss for dense object detection. IEEE Transactions on Pattern Analysis and Machine Intelligence **42**(2), 318–327 (2020). https://doi.org/10.1109/TPAMI.2018.2858826
33. Liu, Y., Ott, M., Goyal, N., et al.: Roberta: a robustly optimized bert pretraining approach. CoRR **abs/1907.11692** (2019), https://arxiv.org/abs/1907.11692
34. Lu, J., Batra, D., Parikh, D., Lee, S.: Vilbert: pretraining task-agnostic visiolinguistic representations for vision-and-language tasks. In: 33rd Conference on Neural Information Processing Systems (2019)
35. Lu, S., Guo, D., Ren, S., Huang, J., et al.: Codexglue: a machine learning benchmark dataset for code understanding and generation. In: Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track. OpenReview.net (2021), url={https://openreview.net/forum?id=6lE4dQXaUcb}
36. Lu, S., Guo, D., Ren, S., et al.: Implementation of codexglue. https://github.com/microsoft/CodeXGLUE (2022), online, accessed on 25 August 2023
37. Mazuera-Rozo, A., Mojica-Hanke, A., Linares-Vásquez, M., Bavota, G.: Shallow or deep? an empirical study on detecting vulnerabilities using deep learning. In: IEEE/ACM 29th International Conference on Program Comprehension. pp. 276–287 (2021). https://doi.org/10.1109/ICPC52881.2021.00034
38. Mendoza, J., Mycroft, J., Milbury, L., Kahani, N., Jaskolka, J.: On the effectiveness of data balancing techniques in the context of ml-based test case prioritization. In: 18th International Conference on Predictive Models and Data Analytics in Software Engineering. p. 72–81. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3558489.3559073
39. Pidgin team: Pidgin website. https://pidgin.im/ (2020), online, accessed on 25 August 2023
40. Pinconschi, E.: Repository of devign on github. https://github.com/epicosy/devign (2020), online, accessed on 25 August 2023
41. Sam Leffler, S.G.: Repository of libtiff on gitlab. https://gitlab.com/libtiff/libtiff (2022), online, accessed on 25 August 2023
42. Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Vats, I., Moazen, H., Sarro, F.: A survey on machine learning techniques for source code analysis. CoRR **abs/2110.09610** (2021), https://arxiv.org/abs/2110.09610
43. Shen, Z., Chen, S., Coppolino, L.: A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. Security and Communication Networks **2020** (January 2020). https://doi.org/10.1155/2020/8858010
44. Shu, R., Xia, T., Williams, L., Menzies, T.: Dazzle: using ooptimized generative adversarial networks to address security data class imbalance issue. In: 19th International Conference on Mining Software Repositories. p. 144–155. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3524842.3528437
45. Sun, C., Myers, A., Vondrick, C., Murphy, K., Schmid, C.: Videobert: a joint model for video and language representation learning. In: IEEE/CVF International Conference on Computer Vision (ICCV). pp. 7463–7472. IEEE Computer Society, Los Alamitos, CA, USA (November 2019). https://doi.org/10.1109/ICCV.2019.00756
46. Truta, C., Randers-Pehrson, G., Dilger, A.E., Schalnat, G.E.: Repository of libpng on github. https://github.com/glennrp/libpng (2023), online, accessed on 25 August 2023
47. Vaswani, A., Shazeer, N., Parmar, N., et al.: Attention is all you need. In: 31st Conference on Neural Information Processing Systems. Curran Associates, Inc. (2017), https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

48. VLC team: Vlc media player website. https://github.com/videolan/vlc (2023), online, accessed on 25 August 2023
49. Wang, S., Liu, W., Wu, J., Cao, L., Meng, Q., Kennedy, P.: Training deep neural networks on imbalanced data sets. In: International Joint Conference on Neural Networks. pp. 4368–4374. IEEE (July 2016). https://doi.org/10.1109/IJCNN.2016.7727770
50. Yang, Z., Shi, J., He, J., Lo, D.: Natural attack for pre-trained models of code. In: International Conference on Software Engineering. p. 1482–1493. Association for Computing Machinery (2022). https://doi.org/10.1145/3510003.3510146
51. You, Y., Zhang, Z., Hsieh, C., Demmel, J.: 100-epoch imagenet training with alexnet in 24 minutes. CoRR **abs/1709.05011** (2017), http://arxiv.org/abs/1709.05011
52. Zhang, H., Li, Z., Li, G., Ma, L., Liu, Y., Jin, Z.: Generating adversarial examples for holding robustness of source code processing models. In: Proceedings of the AAAI Conference on Artificial Intelligence. pp. 1169–1176 (2020). https://doi.org/10.1609/aaai.v34i01.5469
53. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks, p. 10197–10207. Curran Associates Inc., Red Hook, NY, USA (2019)
54. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: 33rd International Conference on Neural Information Processing Systems. pp. 10197–10207. Curran Associates Inc., Red Hook, NY, USA (December 2019), https://dl.acm.org/doi/pdf/10.5555/3454287.3455202
55. Zou, Y., Yu, Z., Vijaya Kumar, B.V.K., Wang, J.: Unsupervised domain adaptation for semantic segmentation via class-balanced self-training. In: European Conference on Computer Vision. pp. 297–313. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-030-01219-9_18