VF-Detector: Making Multi-Granularity Code Changes on Vulnerability Fix Detector Robust to Mislabeled Changes

Zhenkan Fu[1,2] , **Shikai Guo**[1,2] , **Hui Li**[1] , **Rong Chen**[1] , **Xiaochen Li**[3] and **He Jiang**[2,3*]
[1]School of Information Science and Technology, Dalian Maritime University, Dalian, China
[2]Dalian Key Laboratory of Artificial Intelligence, DUT Artificial Intelligence Institute, Dalian, China
[3]School of Software, Dalian University of Technology, Dalian, China
{buildxcb, shikai.guo, li_hui, rchen}@dlmu.edu.cn, {xiaochen.li, jianghe}@dlut.edu.cn

Abstract

As software development projects increasingly rely on open-source software, users face the risk of security vulnerabilities from third-party libraries. To address label and character noise in code changes, we present VF-Detector to automatically identifying bug-fix commits in actual noise development environment. VF-Detector consists of three components: Data Pre-processing (DP), Vulnerability Confidence Computation (VCC) and Confidence Learning Denoising (CLD). The DP component is responsible for preprocessing code change data. The VCC component calculates code change confidence value for each bug-fix by extracting features at various granularity levels. The CLD component removes noise and enhances model robustness by pruning noisy data with confidence values and performing effort-aware adjustments. Experimental results demonstrate VF-Detector's superiority over state-of-the-art methods in EffortCost@L and Popt@L metrics on Java and Python datasets. The improvements were 6.5% and 5% for Java, and 23.4% and 17.8% for Python.

1 Introduction

The growing reliance on third-party libraries in software development has escalated user vulnerability to security threats from these libraries. A notable example is the Heartbleed vulnerability (CVE-2014-0160), a critical bug in the OpenSSL encryption library [Carvalho *et al.*, 2014]. Located in OpenSSL's heartbeat extension, the bug enabled attackers to read protected memory, exposing sensitive information like private keys and passwords. Heartbleed stemmed from lacking boundary checks in *memcpy*(), enabling attackers to exploit OpenSSL's 64KB buffer and leak memory contents.

To address the rising risk of security vulnerabilities in software ecosystems, focus has been on developing Software Composition Analysis (SCA) tools [Zhou and Sharma, 2017; Kang *et al.*, 2022; Imtiaz *et al.*, 2023]. These tools alert users about library vulnerabilities but face a delay in vulnerability disclosure, leaving systems exposed to undetected threats. To identify security-relevant code changes before public disclosure, studies [Nguyen-Truong *et al.*, 2022; Nguyen *et al.*, 2022; Zhou *et al.*, 2021; Nguyen *et al.*, 2023] have developed tools for automatically detecting bug-fix commits using resources like commit messages or issue reports. Zhou et al. [Zhou *et al.*, 2021] used CodeBERT-based deep learning for automated identification and repair of vulnerabilities. To mitigate noise from entangled commits, Nguyen et al. [Nguyen *et al.*, 2023] proposed MiDas, which uses base models for different code granularities in an ensemble model. Despite advancements, challenges in bug-fix detection persist.

Labelling and characterizing noise. Deep learning-based code analysis techniques, reliant on extensive data for optimal training, face inherent challenges during the sample collection process. The large-scale labeling of samples as "positive" or "negative" inevitably leads to label noise, resulting in misclassifications-positive samples marked as negative and vice versa. Additionally, characterization noise can occur through formal code changes that don't alter fundamental semantics. It can also occur through the introduction of new code features, leading to incorrect label classification. Such noise adversely affects the accuracy of these models. Consequently, the key challenge lies in effectively managing data noise to improve data quality.

Lack of robustness and generalization. Code changes contain essential resources for identifying bug-fix commits, yet only a minuscule fraction of these commits are related to vulnerability fixes. Existing approaches [Zhou and Sharma, 2017; Kang *et al.*, 2022; Imtiaz *et al.*, 2023], often trained on specific datasets, risk becoming overly tuned to features of non-vulnerability-related code changes, diminishing their ability to detect bug-fix commits. This leads to reduced robustness and generalization. Insufficient robustness may cause inaccurate judgments and model instability, while limited generalization decreases the model's predictive performance on new data. Therefore, a key challenge is to mitigate the impact of highly imbalanced datasets on model robustness and to enhance the model's effectiveness and robustness.

To tackle identified challenges, we introduce VF-Detector, a multi-granularity vulnerability detector using a confidence learning approach. It comprises three components: Data Pre-processing (DP), Vulnerability Confidence Computation (VCC), and Confidence Learning Denoising (CLD). The DP component is responsible for preprocessing code change data.

*Corresponding Author

The VCC component creates separate neural networks for various code change granularity levels (commit, file, hunk, and line) . It then utilizes an ensemble model to compute sample confidence values for bug-fix in code changes. The CLD component handles label and representational errors by using a VCC-derived confidence matrix and a probability threshold, thus enhancing VF-Detector's ability to recognize bug-fix commits. Subsequently, CLD employs effort-aware adjustments to modify the output probabilities of the neural classifier. This eliminates the adverse effects of irrelevant code changes on VF-Detector's capability to identify bug-fix, thereby improving VF-Detector's robustness.

To validate the effectiveness of VF-Detector, we conducted a series of experiments on code changes extracted from Java and Python open-source projects. The experimental results show that the performance of VF-Detector outperforms the baseline methods in *AUC*, *EffortCost@L* and $P_{opt}@L$, compared with baseline methods, VF-Detector improves the *EffortCost@L* and $P_{opt}@L$ by 6.5% and 5% in Java, respectively, 23.4% and 17.8% in Python. Therefore, VF-Detector is effective in detecting bug-fix commits in code changes and robustly handles noise data in code changes during the training process.

In summary, the contributions of this study are as follows:

- We introduce VF-Detector to detect bug-fix commits, utilizing Data Pre-processing, Vulnerability Confidence Computation, and Confidence Learning Denoising components. VF-Detector makes the Multi-Granularity Code Changes on Vulnerability Fix Detector Robust to mislabeled changes.
- To validate the effectiveness of VF-Detector, we conducted a series of experiments on code changes extracted from Java and Python open-source projects.
- Our source code and datasets are publicly accessible for future research advancements [VF-Detector, 2024].

2 Related Work

The increasing use of third-party libraries in software development underscores the importance of detecting bug-fix commits to safeguard users from security risks. Numerous studies suggesting bug-fix commit detection through commit messages and code changes risk unintentionally incorporating sensitive vulnerability data [Zhou *et al.*, 2022; Nguyen-Truong *et al.*, 2022]. Therefore, focusing on other types of commit-related classification issues and solely on code changesas a dual approach, offering a more secure and effective method for identifying bug-fix commits.

Focusing on other types of commit-related classification issues. VCCFinder [Perl *et al.*, 2015] employed an SVM model incorporating handcrafted features to identify vulnerability-introducing code change commits. DeepCVA [Le *et al.*, 2021] utilized a multi-task learning method with an attention-based Convolutional Gated Recurrent Unit to evaluate the exploitability, impact, and severity of vulnerabilities in code commits. DEPA [Zhong *et al.*, 2022] applied the GRAPA tool to extract vulnerability signatures from previous bug-fix commits for detecting new vulnerabilities in ongoing projects.

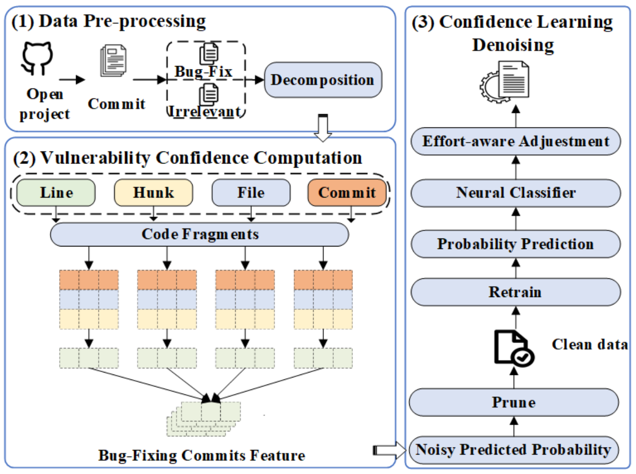

Figure 1: Overall framework of VF-Detector

Solely on code changes. VulFixMiner [Zhou *et al.*, 2021] is a deep learning-based analysis technique that employs CodeBERT for the automated detection of bug-fix commits at the file level. MiDas [Nguyen *et al.*, 2023] employs ensemble learning across multiple code granularity levels for precise identification of fix-related messages.

3 VF-Detector Model

3.1 Overview

In this section, we provide a comprehensive description of the VF-Detector framework, as illustrated in the Figure 1. VF-Detector comprises three components: DP, VCC and CLD.

Firstly, DP component segregates code changes from Java and Python open-source projects into bug-fix commits and unrelated commits. Then, multi-layered information from code changes is extracted at several granularity levels (line, hunk, file, and commit). This extracted information is then encoded into numerical vectors through code embeddings, serving as input for the feature extraction layer of the VCC component. Secondly, the feature extraction layer of VCC extracts features from code commits at each granularity level. It uses a neural classifier to calculate the confidence values of code change samples in bug-fix by learning the mapping from the final representation of the input to the corresponding output vector. Thirdly, the CLD component utilizes the confidence matrix from VCC, along with code change labels, to estimate samples within the confidence joint, pruning noisy data. Finally, using processed code change data for retraining, it identifies and learns the features of bug-fix commits in code changes. CLD employs effort-aware adjustments to modify the output probabilities of the neural classifier, thereby enhancing the robustness of the VF-Detector.

3.2 Vulnerability Confidence Computation

Code Decomposition and Code Embedding

In terms of the structure of a code commit, a commit consists of a set of code changes applied to a group of files. Each file is composed of multiple code blocks located in specific regions, which are a series of code changes applied to Lines of

Code (LOC). VF-Detector followes the natural organizational structure of a commit, decomposes the commit into four granularity levels of code fragments (commit, file, hunk and line). For instance, at the line-level granularity, VCC divides the code changes into numerous committed lines, incorporating each input commit line into a sequence of LOCs.

Subsequently, the extracted code fragments need to be encoded into high-dimensional vectors. VF-Detector finetunes CodeBERT separately at each granularity level [Feng *et al.*, 2020; Yang *et al.*, 2022; Mashhadi and Hemmati, 2021; Xia and Zhang, 2022], to capture the specific features of code changes at each level. The finetuned models are then used as code embedding models to represent the code fragments. By default, CodeBERT takes two segments as input: one from natural language (NL) and the other from programming language (PL). The format of its input is

$$[CLS]\langle NL\rangle[SEP]\langle PL\rangle[EOS] \tag{1}$$

In Equal (1), the tokens denoted as *[CLS]*, *[SEP]*, and *[EOS]* are considered as specialized tokens within the context of CodeBERT. In particular, the token denoted as *[CLS]* signifies the initiation of the CodeBERT sequence, which is subsequently followed by natural language text. The *[SEP]* token separates natural language text from programming language source code, and the *[EOS]* token marks the end of the CodeBERT sequence. CodeBERT is pretrained for two distinct data modalities: bimodal data (pairs of natural language and source code) and unimodal data (source code only). Therefore, in VF-Detector, special attention is given to the code added and deleted in the input commit, considering the presence of source code context in two different ways.

Context-dependent Representation. In this representation, the deleted code and the code added in the code fragment are treated as a pair of data. This method aims to learn the joint representation of code added and deleted in a commit. This approach contextualizes added code in relation to deleted code, and the reverse is also true. More formally, a code fragment is represented in CodeBERT's input format as

$$[CLS]\langle rem-code\rangle[SEP]\langle add-code\rangle[EOS] \tag{2}$$

Next, VCC forwards this representation to the CodeBERT model and use the output at the *[CLS]* token as the initial embedding of the code fragment.

Context-free Representation. In this representation, deleted code and added code in the code fragment are treated as two distinct unimodal data points. This method separately processes deleted and added code without considering their corresponding counterparts. More formally, a code fragment is represented in CodeBERT's input format as

$$[CLS]\langle empty\rangle[SEP]\langle add-code\rangle[EOS] \tag{3}$$

$$[CLS]\langle empty\rangle[SEP]\langle rem-code\rangle[EOS] \tag{4}$$

Then, VCC component forwards these representations to the CodeBERT model. In this scenario, VCC acquires two initial embeddings for the code fragment: one for the added code and another for the deleted code. It's noteworthy that CodeBERT can process up to a maximum of 512 tokens. Hence, if the input surpasses this limit, VCC component considers only the first 512 tokens for truncation. By integrating the granularity of four levels with two distinct types of code fragments, VCC obtains seven configurations for commit embeddings, as illustrated in Table 1.

Representation	Granularity	Feature Extrator
Context-dependent	Commit	FCN
Context-dependent	File	FCN
Context-dependent	Hunk	CNN
Context-free	Commit	FCN
Context-free	File	FCN
Context-free	Hunk	CNN
Context-free	Line	LSTM

Table 1: Commit Embedding Settings

Feature Extraction

VF-Detector observed that features at the four granularity levels are distinct. Therefore, to effectively extract features from each model, VCC employs different models accordingly. Overall, feature extraction at each granularity level follows a common structure, including a feature extractor and a feature fusion layer. It is important to note that each base model has a feature extractor and a feature fusion layer. The design of the feature extractor is customized according to each granularity, while the feature fusion layer is shared across different granularities. Below we detail these steps.

(1) Feature Extractor. VF-Detector utilizes four deep learning models as feature extractors for different granularity levels. For each commit, the feature extractor corresponding to a specific granularity takes the embedding vector as input and returns a feature vector as output. We will present the detailed architectures of these models.

(1.a) Line-Level. Given that lines in code changes are read sequentially, VCC leverages a Recurrent Neural Network (RNN), a standard model for processing sequential data. This is used to extract features at the line-level granularity. VCC interprets code changes as a sequence of lines, each represented by an embedding vector, with the sequence denoted by $[l_1, l_2, ...l_{T_{line}}]$. VCC then employs a Bi-directional Long Short-Term Memory (BiLSTM) model as our feature extractor. In our case, the BiLSTM utilizes a forward LSTM to read the commit from l_1 to $l_{T_{line}}$, and a backward LSTM to read the commit from $l_{T_{line}}$ to l_1. VCC obtains the final output of the LSTM as the characteristics of the commit.

$$f_{line} = BiLSTM([l_1, l_2, ...l_{T_{line}}]) \tag{5}$$

(1.b) Hunk-Level. Unlike lines, large blocks in a commit do not carry sequential relationships. However, there still exist dependencies between adjacent blocks, such as blocks within the same file. Dependencies may encompass shared variables, constants, as well as function calls. Therefore, VCC utilizes a Convolutional Neural Network (CNN), which has proven its ability to capture local dependencies in many tasks. Specifically, for a set of embedding vectors $[h_1, h_2, ...h_H]$ from block-level code fragments in a commit, VCC aggregates information using convolution layers on adjacent blocks. More formally, the features of the *i-th* hunk-level embedding vector are represented by aggregating information from neighboring embedding vectors:

$$h'_i = Conv([h'_{i-(w-1)/2}, ...h'_{i+(w-1)/2}]) \tag{6}$$

where *w* represents the kernel size of the convolutional filters in the convolutional layer. Following this, a max-pooling layer extracts key features from the input embeddings, yielding the final feature representation.

$$f_{hunk} = MaxPool([h'_1, h'_2, ...h'_H]) \tag{7}$$

(1.c) File-Level. At the file level, VF-Dector is to capture the high-level relationships of all code in a commit. Hence, VCC uses a Fully Connected Neural Network (FCN) to simultaneously capture relationships among all files in the commit. Specifically, for file-level code fragments in a commit represented by embedding vectors $[f_1, f_2, ...f_F]$, VCC represents the commit by concatenating the features of all vectors from f_1 to f_F. Consequently, VCC obtains an $n \times F$ dimensional vector representing the commit, where n is the vector dimension of each file (i.e., the output size of CodeBERT). It's important to note that fully connected layers typically require input features of a fixed size. Therefore, to address this issue, VCC set F as a predefined parameter. For each commit, if its number of files is less than F, VCC adds some empty files to ensure all commits have the same number of files. Otherwise, VCC truncates it, retaining only its first F files. After obtaining a fixed-size input vector, VCC uses a fully connected layer to acquire the output feature as follows:

$$f_{file} = FCN(f_1 \oplus f_2 \oplus ... \oplus f_F) \tag{8}$$

where $\oplus$ denotes the concatenation operator, and FCN is a fully connected layer with an input size of $n \times F$.

(1.d) Commit-Level. Similar to the feature extraction at the file level, VCC component also uses a fully connected layer for the commit level. Specifically, given x as the commit-level embedding vector generated by CodeBERT for a given commit. VCC employs a fully connected layer to obtain the output feature, as follows:

$$f_{commit} = FCN(x) \tag{9}$$

where FCN is a fully connected layer with an input size equal to the size of x, and the output size of CodeBERT is n.

(2) Feature Fusion. On top of the extracted feature vectors, a set of fully connected layers are constructed for feature fusion. VF-Detector employs two different types of feature : bimodal and unimodal fusion.

(2.a) Bimodal Fusion. For the bimodal representation, VCC obtains a single feature vector and directly input it into a linear layer for feature fusion.

(2.b) Unimodal Fusion. VCC has two feature vectors, one for added code and the other for deleted code. Hence, VCC first concatenates them into one vector, and then input the vector into a linear layer to fuse the features.

Finally, VCC uses a neural network classifier in conjunction with multi-level commit features to calculate the confidence values of code change samples applied to bug-fix. This confidence value is then used as input in Section 3.3.

3.3 Confidence Learning Denoising

Data Denoising

To mitigate the issue of feature noise and label errors in code change, we introduce the CLD component to address noise in bug-fix commit detection. This component is based on the Confident Learning (CL) framework [Northcutt *et al.*, 2021].

In a set of code change commit samples contaminated with noisy data, CLD component denotes the set of binary problem labels by $[m]$. Label 0 indicates that the sample is unrelated to bug-fix, and label 1 signifies that the sample is a bug-fix commit. Let X represent collection of n instances with observed noisy labels $\tilde{y}$, i.e., $X = (x, \tilde{y})^n \in (R^d, [m])^n$, where x denotes a data sample. The pair $(x, \tilde{y})$ represents the noisy data requiring cleansing in samples X. For each data instance, CLD component assumes the existence of a latent true label y^*. In the code change X, the subset of instances i with noisy labels is denoted as $X_{\tilde{y}=i}$. For each data sample x,$\hat{p}(\tilde{y} = i, x \in X_{\tilde{y}=i}, \theta)$ represents the confidence value of the corresponding label $\tilde{y}$. The lower the confidence, the higher the likelihood of label inaccuracy.

The CLD component of the VF-Detector method requires two inputs: (1) a confidence matrix $\hat{P}_{k,i} = \hat{p}\left(\tilde{y} = I; x_k, \theta\right)$, containing the predicted probabilities of each sample for different labels by the VCC component. (2) A vector $\tilde{y}_k, x_k \in X$ of noisy labels, representing the labels for each sample in the code change. These two inputs are linked via index k and are applicable for all $x_k \in X$.

The CLD component enhances learning by identifying representational noise and label errors in the code change, employing a three-step procedure for data refinement.

(1) Count. This step describes the joint distribution of the noisy label $\tilde{y}$ and the true label y^*. CLD estimates the joint distribution matrix $\hat{Q}_{\tilde{y},y^*}$ to characterize class-conditional label noise. For each instance in the code change, CLD computes the likelihood of it being a bug-fix commit and calibrate accordingly. In the confidence joint matrix $C_{\tilde{y},y^*}$, a high probability value $\hat{p}\left(\tilde{y} = j; x, \theta\right)$ for a sample x with label $\tilde{y} = j$ indicates a potential belonging to $y^* = j$. The diagonal entries of $C_{\tilde{y},y^*}$ count correct labels, while off-diagonal entries capture asymmetric label error counts. The representation of the confidence joint matrix is as follows:

$$\begin{aligned} C_{\tilde{y},y*} &:= \{x \in X\tilde{y} = i : \hat{p}\left(\tilde{y} = j; x, \theta\right)\} \geqslant t_j, \\ j &= \underset{l \in [m]: \hat{p}(\tilde{y}=l; x,\theta)\} \geqslant t_j}{\arg\max} \hat{p}\left(\tilde{y} = l; x, \theta\right)\} \end{aligned} \tag{10}$$

where the threshold t_j for each category represents the expected confidence level for that particular category.

$$t_j = \frac{1}{|X_{\tilde{y}=j}|} \sum_{x \in X_{\tilde{y}=j}} \hat{p}(\tilde{y} = j; x, \theta) \tag{11}$$

These thresholds enhance the robustness of the CLD component in quantifying uncertainty. They address two key aspects: heterogeneous category probability distributions and imbalance in category quantities. To illustrate with a practical scenario, in code change commits, only a minuscule portion is pertinent to bug-fix. Here, non-bug-fix samples often show higher probabilities due to VF-Detector's excessive confidence in them, resulting in a corresponding increase in their threshold values. Conversely, the threshold for bug-fix commits may be lower or even zero, meaning no processing for bug-fix commit samples. The existence of these thresholds allows us to conjecture about y^*, despite the high category imbalance, thus avoiding overconfident guesses about y^*. The confidence joint matrix $C_{\tilde{y},y^*}$ not only performs well in anomaly detection but also offers considerable flexibility in threshold selection. CLD estimates the label noise in code change X using the confidence joint matrix $C_{\tilde{y},y^*}$, which characterizes the joint distribution matrix $\hat{Q}_{\tilde{y},y^*}$.

$$\hat{Q}_{\tilde{y}=i,y^*=j} = \frac{\frac{C_{\tilde{y}=i,y^*=j}}{\sum_{j\in M} C_{\tilde{y}=i,y^*=j}} \cdot |X_{\tilde{y}=i}|}{\sum_{i\in M, j\in M} \left(\frac{C_{\tilde{y}=i,y^*=j}}{\sum_{j\in M} C_{\tilde{y}=i,y^*=j}} \cdot |X_{\tilde{y}=i}| \right)} \tag{12}$$

The primary objective of estimating the joint distribution is to detect and eliminate representational noise and label inaccuracies in subsequent processes. The joint distribution effectively reflects real-world error and true label distributions [Northcutt *et al.*, 2021]. As sample size grows, estimations become more precise, closely mirroring the true distribution.

(2) Cleansing. CLD utilizes probabilistic ranking to clean the training data after joint estimation, pruning, and sorting. Given that identifying bug-fix commits is a binary classification task with few actual bug-fix commits, our focus is on estimating and cleansing incorrect labels. This specifically targets the negative samples.

(3) Re-training. Utilizing the aforementioned methods, CLD filters out representational noise and label inaccuracies. The denoised code changes is then fed into the VCC for re-training, thereby identifying and learning the characteristics of bug-fix commits in code changes.

Therefore, the CLD component addresses the challenge of label and character noise in code changes.

Effort-Aware Adjustment

To increase the detection of bug-fix commits within a limited inspection cost, i.e., Lines of Code (LOC) checked, the CLD component employ effort-aware adjustment. This adjustment involves modifying the output of the neural network classifier based on the length of the commit, giving priority to shorter bug-fix commits over longer ones. Specifically, our effort-aware adjustment is defined as follows:

$$P(c) = prob_c \times f(loc_c) \tag{13}$$

In Equal (13), $P(c)$ represents the modification applied to the predicted probabilities by a neural classifier for a specific commit c denoted as $prob_c$. VF-Detector aims for $P(c)$ to be directly proportional to the count of LOC in c, loc_c. The higher the count of LOC, the greater the adjustment magnitude. Hence, VF-Detector defines $f(loc_c)$ as a function characterizing this trajectory. However, it is crucial that $f(loc_c)$ be meticulously designed to ensure that the adjustment does not overshadow the probabilities predicted by the neural classifier. To this end, VF-Detector opt for a logarithmic function as our choice for f. More formally,

$$f(loc_c) = \log_a(loc_c) \tag{14}$$

Datasets	Lang.	bug-fix Commits				Non-bug-fix Commits				Project
		C	F	H	L	C	F	H	L	Number
Train	Java	983	2,011	7,205	35,423	31,323	74,661	281,656	1,314,231	120
	Python	522	747	2,124	8,769	20,362	27,737	75,618	294,982	84
Validation	Java	191	224	798	3,801	6,921	8,296	31,106	147,286	119
	Python	80	83	240	916	2,949	3,082	8,744	32,450	83
Test	Java	300	689	2,522	11,346	87,856	208,363	859,385	3,670,328	30
	Python	195	254	613	2,384	55,638	72,752	205,763	784,006	22

Table 2: Dataset Details

In the code change data, a represents the maximum number of LOC in bug-fix commits. Given that the value of loc_c is greater than or equal to 1 and less than a, the bounds of $\log_a(loc_c)$ for any commit within the code changes range from 0 to 1. Therefore, VF-Detector suggests a revised definition for effort-aware adjustment:

$$P(c) = prob_c \times \log_a(loc_c) \tag{15}$$

Utilizing the proposed effort-aware adjustment approach, the output probabilities of the neural classifier were recalibrated to yield the final scores for each commit as follows:

$$S(c) = prob_c - P(c) \tag{16}$$

where c denotes a given commit, $prob_c$ represents the output probability from the neural classifier, and $P(c)$ is the computed value of c under effort-aware adjustment. However, in real-world scenarios, there might be bug-fix commits with lengths exceeding a, leading to a negative $S(c)$ as Equal (16). Given our preference for shorter commits during inspection, these larger commits would be ranked poorly; these outliers are disregarded. To maintain the integrity of our evaluation, VF-Detector constrain $S(c)$ to a minimum of zero as follows:

$$S(c) = \max(prob_c - P(c), 0) \tag{17}$$

In conclusion, effort-aware adjustment will modify the predicted probabilities of all commits in code changes. This eliminates the adverse effects of irrelevant code changes on the model's recognition of vulnerability fixes. Thereby, it enhances the robustness of VF-Detector.

4 Experimental Setting

4.1 Datasets

For comparison with the baseline, we evaluated VF-Detector using the dataset proposed by VulFixMiner and SAP [Zhou *et al.*, 2021; Ponta *et al.*, 2019]. This dataset comprises bug-fix and non-bug-fix commits from 150 Java and 106 Python open-source projects. The Java dataset contains 1,436 bug-fix commits and 474,555 non-bug-fix commits, while the Python dataset includes 885 bug-fix commits and 357,696 non-bug-fix commits. The dataset is split into training and testing sets in an 80%/20% ratio, with 80% of the training set used for training and 20% for validation. To reduce data imbalance, random undersampling was performed on the training and validation sets. Table 2 details the dataset composition at commit (C), file (F), hunk (H), and line (L) levels.

4.2 Baselines

VulFixMiner and MiDas [Zhou *et al.*, 2021; Nguyen *et al.*, 2023] are leading baselines in bug-fix commit detection, employing code change-based methods for vulnerability detection. The distinction lies in VulFixMiner extracting code commits at a file-level granularity, while MiDas considers commits at four different granularities. DeepJIT [Hoang *et al.*, 2019] is a defect prediction method using deep learning, analyzing code changes and commit information to predict the defectiveness of a commit. LApredict [Zeng *et al.*, 2021] is a method using logistic regression and a single feature that excels in detecting defective program changes. Additionally, there is a simple baseline model, the LOC-Sensitive Model, based on the number of Lines of Code changed. It ranks commits in ascending order by line count to detect more bug-fix commits under a fixed line count threshold. We replicate their techniques on the Java and Python code changes dataset when the papers offer source code (VulFixMiner, MiDas, DeepJIT, LApredict, and LOC-Sensitive Model).

4.3 Evaluation Metrics

For a fair assessment, we follow the evaluation metrics from prior vulnerability detection research [Nguyen *et al.*, 2023], using *AUC*, *EffortCost@L*, and *Popt@L* as our key criteria. The value of L is set at 5%, 10%, 15%, and 20%.

4.4 Training Details

VF-Detector's foundational models, blending CodeBERT with a feature extractor, are independently trained for diverse commit classifications. Employing the Adam optimizer [Kingma and Ba, 2015] with a learning rate of 1e-5, the training culminates in refining the neural classifier to integrate outputs for prediction. The detailed training parameters and other specifics are defined on GitHub [VF-Detector, 2024].

5 Experimental Results

5.1 Performance Compared to Baselines

Motivation. The experiment compares VF-Detector's ability to identify bug-fix commits with existing methods.

Results. The comparison results of VF-Detector with baselines are shown in Table 3. On the Java dataset, VF-Detector is found to be slightly less effective in *AUC* compared to MiDas, yet it surpasses the baselines in *CostEffort* and P_{opt} metrics. The improvement ranges from 6.3% to 6.5% in *CostEffort* and from 2.7% to 5% in P_{opt} as the total LOC inspected increases from 5% to 20%. At 20% LOC, VF-Detector can identify over 90% of bug-fix commits. On the Python dataset, our model, VF-Detector, outperforms the best baseline in all metrics. For effort-aware metrics, VF-Detector shows an increase of 8.6% to 23.4% in *CostEffort* and 12.1% to 17.8% in P_{opt}. From all these results, VF-Detector demonstrates higher discriminative capability in identifying vulnerability-fixing commits, able to detect more such commits at the same inspection cost.

5.2 Performance Affect by Training Set Size

Motivation. The size of the training set directly influences the learning capability of a neural network. The aim of this experiment is to evaluate the impact of varying dataset sizes on the performance of VF-Detector in identifying bug-fix commits. We tested the performance of VF-Detector on training sets comprising 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% and 100% of the total dataset.

Lang.	Model	AUC	CostEffort				P_{opt}			
			5%	10%	15%	20%	5%	10%	15%	20%
Java	VulFixMiner	0.81	0.61	0.65	0.68	0.71	0.53	0.58	0.61	0.63
	Midas	**0.85**	0.64	0.77	0.87	0.91	0.50	0.60	0.67	0.73
	DeepJIT	0.83	0.34	0.48	0.50	0.62	0.24	0.33	0.38	0.43
	LApredict	0.45	0.22	0.38	0.49	0.59	0.13	0.21	0.29	0.35
	LOC-sensitive model	0.37	0.32	0.50	0.59	0.67	0.19	0.30	0.39	0.45
	VF-Detector	0.80	**0.68**	**0.82**	**0.87**	**0.91**	**0.50**	**0.63**	**0.70**	**0.75**
Python	VulFixMiner	0.73	0.32	0.40	0.48	0.56	0.24	0.30	0.35	0.39
	Midas	0.83	0.47	0.64	0.74	0.81	0.33	0.45	0.53	0.59
	DeepJIT	0.60	0.08	0.13	0.22	0.33	0.05	0.08	0.12	0.16
	LApredict	0.48	0.12	0.17	0.23	0.29	0.08	0.11	0.14	0.17
	LOC-sensitive model	0.47	0.27	0.44	0.52	0.61	0.16	0.25	0.33	0.39
	VF-Detector	**0.83**	**0.58**	**0.76**	**0.85**	**0.88**	**0.37**	**0.53**	**0.62**	**0.68**

Table 3: Comparison with Baselines

Lang.	Dataset	AUC	CostEffort				P_{opt}			
	Size		5%	10%	15%	20%	5%	10%	15%	20%
Java	10%	0.51	0.37	0.55	0.65	0.71	0.23	0.36	0.45	0.50
	20%	0.56	0.44	0.60	0.69	0.76	0.29	0.41	0.49	0.55
	30%	0.62	0.50	0.67	0.77	0.82	0.35	0.48	0.56	0.62
	40%	0.64	0.56	0.69	0.77	0.84	0.38	0.51	0.58	0.64
	50%	0.70	0.61	0.74	0.81	0.85	0.43	0.56	0.64	0.68
	60%	0.74	0.63	0.76	0.83	0.89	0.44	0.58	0.65	0.70
	70%	0.76	0.68	0.80	0.83	0.88	0.46	0.60	0.68	0.72
	80%	0.78	0.65	0.81	0.85	0.90	0.47	0.61	0.69	0.73
	90%	0.79	0.65	0.79	0.86	0.89	0.48	0.61	0.68	0.73
	100%	**0.80**	**0.68**	**0.82**	**0.87**	**0.91**	**0.50**	**0.63**	**0.70**	**0.75**
Python	10%	0.56	0.34	0.49	0.62	0.70	0.21	0.31	0.39	0.46
	20%	0.63	0.39	0.57	0.70	0.75	0.28	0.40	0.48	0.54
	30%	0.70	0.48	0.68	0.77	0.82	0.35	0.48	0.56	0.62
	40%	0.72	0.51	0.66	0.75	0.84	0.37	0.48	0.56	0.62
	50%	0.76	0.56	0.71	0.78	0.84	**0.38**	0.51	0.59	0.64
	60%	0.79	**0.59**	0.73	0.82	0.87	0.36	0.52	0.61	0.67
	70%	0.81	0.58	0.76	0.83	0.88	0.37	0.53	0.62	0.68
	80%	0.81	0.57	0.76	0.84	0.87	0.37	0.53	0.62	0.68
	90%	0.81	0.56	0.72	0.84	0.86	0.35	0.51	0.60	0.66
	100%	**0.83**	0.58	**0.76**	**0.85**	**0.88**	0.37	**0.53**	**0.62**	**0.68**

Table 4: VF-Detector's Performance as the Dataset Size Varies

Results. Table 4 presents the performance results of VF-Detector as the dataset size varies. When the dataset increases from 10% to 100%, in the case of Java, AUC improves from 0.51 to 0.8, and the highest improvements in CostEffort and P_{opt} are 6.15% to 18.92% and 10% to 26.09%. For Python, AUC increases from 0.56 to 0.83, and the highest improvements in CostEffort and P_{opt} are 9.33% to 23.08% and 17.39% to 33.33%. In summary, the training dataset size impacts the performance of VF-Detector in identifying bug-fix. With an increase in the training dataset, VF-Detector's learning capability improves, enhancing its bug-fix identification performance. Moreover, when the training dataset is relatively small, the learning capability improves more rapidly with dataset expansion.

5.3 Performance Impact by Noise Reduction

Motivation. Code changes unrelated to bug-fix in code modifications can generate a substantial amount of noise, which significantly impacts the performance of VF-Detector. The CLD component addresses this issue by employing joint estimation, pruning, and sorting of noisy data. The purpose of this experiment is to evaluate the impact of the amount of noise data processed on the performance of VF-Detector. We controlled the proportion of noise data removal at 5%, 7%, 10%, 15%, and 20%.

Results. The results in Table 5 indicate that as the noise reduction ratio increases from 5% to 10%, all performance

Lang	Noise Reduction	AUC	CostEffort				P_{opt}			
			5%	10%	15%	20%	5%	10%	15%	20%
Java	5%	0.8	0.62	0.82	0.87	**0.92**	0.47	0.61	0.69	0.74
	7%	0.8	0.66	0.82	0.86	0.90	0.48	0.60	0.69	0.74
	10%	**0.80**	**0.68**	**0.82**	**0.87**	0.91	**0.50**	**0.63**	**0.70**	**0.75**
	15%	0.79	0.67	0.81	0.86	0.90	0.49	0.63	0.70	0.74
	20%	0.79	0.68	0.8	0.85	0.89	0.48	0.62	0.69	0.73
Python	5%	0.83	0.55	0.75	0.83	0.86	0.34	0.51	0.60	0.66
	7%	0.82	0.51	0.72	0.82	0.86	0.34	0.50	0.59	0.65
	10%	**0.83**	**0.58**	**0.76**	**0.85**	**0.88**	**0.37**	**0.53**	**0.62**	**0.68**
	15%	0.81	0.56	0.77	0.82	0.87	0.35	0.52	0.62	0.68
	20%	0.81	0.57	0.74	0.83	0.85	0.32	0.52	0.61	0.67

Table 5: VF-Detector Performance with Controlled Noise Removal

Lang	Model	AUC	CostEffort				P_{opt}			
			5%	10%	15%	20%	5%	10%	15%	20%
Java	VF-Detector$_{NoAdj}$	**0.85**	0.57	0.71	0.79	0.82	0.43	0.53	0.61	0.66
	VF-Detector	0.80	**0.68**	**0.82**	**0.87**	**0.91**	**0.50**	**0.63**	**0.70**	**0.75**
Python	VF-Detector$_{NoAdj}$	**0.86**	0.45	0.67	0.73	0.80	0.30	0.43	0.52	0.59
	VF-Detector	0.83	**0.58**	**0.76**	**0.85**	**0.88**	**0.37**	**0.53**	**0.62**	**0.68**

Table 6: Performance with and without Effort-Aware Adjustment

metrics except *AUC* show a gradual improvement. In the case of Java's *CostEffort*, there is a 9.68% improvement when L is at 5%, and P_{opt} shows an improvement ranging from 1.35% to 6.38%. For Python, respectively, *CostEffort* and P_{opt} exhibit more significant improvements, ranging from 1.33% to 13.73% and from 3.03% to 8.82%. However, when the noise reduction ratio increases from 10% to 20%, performance metrics begin to show a slight decline. In the case of Python, P_{opt} even decreases by 5%. This suggests that when the noise reduction ratio is low, VF-Detector's performance is susceptible to noise interference. When the noise reduction ratio is too high, VF-Detector's ability to identify bug-fix commits decreases due to the reduction in samples. Overall performance follows a normal distribution with the best performance observed at a 10% noise reduction ratio.

5.4 Performance of Effort-Aware Adjustment

Motivation. Effort-Aware Adjustment Affects the Predictive Capability of the Model. To address this research question, we compared two versions of VF-Detector, one with and the other without (NoAdj) the effort-aware objective function.

Results. Table 6 reveals that VF-Detector, post effort-aware adjustment, saw minor *AUC* reductions of 5.88% in Java and 3.49% in Python. However, there was a significant increase in the *CostEffort* and P_{opt} metrics. Specifically, for Java, the improvement ranged from 10.13% to 19.3% and 13.64% to 18.87%, for Python, it was from 10% to 28.87% and 13.64% to 23.26%. Therefore, effort-aware adjustment can increase the number of commits inspected without compromising VF-Detector's ability to distinguish bug-fix commits. The effort-aware adjustment increases the number of predetermined bug-fix commits at a specific cost in LOC.

5.5 Performance of Different Levels Granularity

Motivation. Previous studies primarily focused on the performance of models in inspecting bug-fix commits within code changes from a single granularity level, namely the file level. This experiment aims to explore the impact of different granularity levels on the performance of VF-Detector. In this experiment, we compared the performance of VF-Detector across various combinations of granularity levels.

Lang	Model	AUC	CostEffort				P_{opt}			
			5%	10%	15%	20%	5%	10%	15%	20%
Java	VF-Detector(C)	0.80	0.53	0.66	0.77	0.84	0.38	0.50	0.57	0.63
	VF-Detector(F)	0.82	0.55	0.66	0.78	0.86	0.43	0.52	0.59	0.65
	VF-Detector(H)	**0.83**	0.59	0.75	0.83	0.90	0.46	0.57	0.64	0.70
	VF-Detector(L)	0.80	0.61	0.73	0.79	0.82	0.46	0.57	0.63	0.68
	VF-Detector(L+H)	0.70	0.60	0.74	0.82	0.86	0.44	0.56	0.63	0.69
	VF-Detector(L+H+F)	0.75	0.65	0.78	0.82	0.89	0.49	0.61	0.67	0.72
	VF-Detector(L+H+F+C)	0.80	**0.68**	**0.82**	**0.87**	**0.91**	**0.50**	**0.63**	**0.70**	**0.75**
Python	VF-Detector(C)	0.80	0.39	0.53	0.65	0.73	0.25	0.36	0.44	0.50
	VF-Detector(F)	0.81	0.44	0.54	0.64	0.72	0.30	0.40	0.46	0.51
	VF-Detector(H)	0.80	0.45	0.64	0.71	0.77	0.29	0.42	0.51	0.56
	VF-Detector(L)	0.82	0.49	0.66	0.75	0.81	0.28	0.43	0.52	0.59
	VF-Detector(L+H0	0.74	0.54	0.69	0.81	0.85	0.36	0.50	0.58	0.64
	VF-Detector(L+H+F)	0.79	**0.59**	0.74	0.83	0.86	**0.41**	0.54	0.63	0.68
	VF-Detector(L+H+F+C)	**0.83**	0.58	**0.76**	**0.85**	**0.88**	0.37	**0.53**	**0.62**	**0.68**

Table 7: VF-Detector's Granularity Level Performance

Results. According to the comparison results in Table 7, it is evident that none of the single-granularity levels of VF-Detector outperforms the others across all performance metrics. Additionally, while the differences in *AUC* are minimal among different granularities of VF-Detector, the single-granularity versions exhibit significantly lower performance. This lower performance is observed in terms of *Cost-Effort* and P_{opt} when compared to the multi-granularity versions. Specifically, in comparison to the multi-granularity VF-Detector, the Java dataset witnesses a maximum reduction of 16% in *CostEffort* and 13% in P_{opt}. The Python dataset shows even more substantial reductions, reaching 25% in *CostEffort* and 18% in P_{opt}. When the Hunk level is introduced, there is a slight decrease in *AUC*. In the case of Java, *CostEffort* and P_{opt} performance is weaker at lower L values but shows improvements of 1.37% to 4.88% as L increases. For Python, there are significant enhancements in *CostEffort* and P_{opt}, with increases ranging from 4.55% to 10.2% and 8.47% to 28.58%. Furthermore, as granularity is further increased to the File level, there is a slight improvement of 0.05 in *AUC*, along with additional enhancements in *CostEffort* and P_{opt}. Specifically, in the Java dataset, these improvements range from 3.49% to 7.69% in *CostEffort* and 4.35% to 11.36% in P_{opt}. In the Python dataset, there are enhancements of 1.18% to 9.26% for *CostEffort* and 6.25% to 18% for P_{opt}. In summary, integrating code change information at different granularities contributes to VF-Detector achieving optimal performance.

6 Conclusion

This paper presents the VF-Detector model, comprising three main components: Data Pre-processing, Vulnerability Confidence Computation and Confident Learning De-noising. VF-Detector addresses the challenges of label and character noise in code changes and model lack of robustness by using code changes confidences matrix to prune noisy data and implementing effort-aware adjustment. Experimental results show that VF-Detector's performance outperforms the baseline methods in *EffortCost@L* and $P_{opt}@L$, improving 6.5% and 5% in Java, respectively, 23.4% and 17.8% in Python.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No.62202079, No.62032004), the Dalian Excellent Young Project (2022RY35).

References

[Carvalho *et al.*, 2014] Marco M. Carvalho, Jared DeMott, Richard Ford, and David A. Wheeler. Heartbleed 101. *IEEE Secur. Priv.*, 12(4):63–67, 2014.

[Feng *et al.*, 2020] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020.

[Hoang *et al.*, 2019] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc, editors, *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pages 34–45. IEEE / ACM, 2019.

[Imtiaz *et al.*, 2023] Nasif Imtiaz, Aniqa Khanom, and Laurie A. Williams. Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages. *IEEE Trans. Software Eng.*, 49(4):1540–1560, 2023.

[Kang *et al.*, 2022] Hong Jin Kang, Truong Giang Nguyen, Xuan-Bach Dinh Le, Corina S. Pasareanu, and David Lo. Test mimicry to assess the exploitability of library vulnerabilities. In Sukyoung Ryu and Yannis Smaragdakis, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 276–288. ACM, 2022.

[Kingma and Ba, 2015] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[Le *et al.*, 2021] Triet Huynh Minh Le, David Hin, Roland Croft, and Muhammad Ali Babar. Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 717–729. IEEE, 2021.

[Mashhadi and Hemmati, 2021] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, pages 505–509. IEEE, 2021.

[Nguyen *et al.*, 2022] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Xuan-Bach Dinh Le, and David Lo. Vulcurator: a vulnerability-fixing commit detector. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 1726–1730. ACM, 2022.

[Nguyen *et al.*, 2023] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Chengran Yang, Zhipeng Zhao, Bowen Xu, Jiayuan Zhou, Xin Xia, Ahmed E. Hassan, Xuan-Bach Dinh Le, and David Lo. Multi-granularity detector for vulnerability fixes. *IEEE Trans. Software Eng.*, 49(8):4035–4057, 2023.

[Nguyen-Truong *et al.*, 2022] Giang Nguyen-Truong, Hong Jin Kang, David Lo, Abhishek Sharma, Andrew E. Santosa, Asankhaya Sharma, and Ming Yi Ang. HERMES: using commit-issue linking to detect vulnerability-fixing commits. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*, pages 51–62. IEEE, 2022.

[Northcutt *et al.*, 2021] Curtis G. Northcutt, Lu Jiang, and Isaac L. Chuang. Confident learning: Estimating uncertainty in dataset labels. *J. Artif. Intell. Res.*, 70:1373–1411, 2021.

[Perl *et al.*, 2015] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 426–437. ACM, 2015.

[Ponta *et al.*, 2019] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc, editors, *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pages 383–387. IEEE / ACM, 2019.

[VF-Detector, 2024] VF-Detector. Our replication package. https://github.com/buildcb/VF-Detector, 2024.

[Xia and Zhang, 2022] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations* *of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 959–971. ACM, 2022.

[Yang *et al.*, 2022] Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural attack for pre-trained models of code. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1482–1493. ACM, 2022.

[Zeng *et al.*, 2021] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. Deep just-in-time defect prediction: how far are we? In Cristian Cadar and Xiangyu Zhang, editors, *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, pages 427–438. ACM, 2021.

[Zhong *et al.*, 2022] Hao Zhong, Xiaoyin Wang, and Hong Mei. Inferring bug signatures to detect real bugs. *IEEE Trans. Software Eng.*, 48(2):571–584, 2022.

[Zhou and Sharma, 2017] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 914–919. ACM, 2017.

[Zhou *et al.*, 2021] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E. Hassan. Finding A needle in a haystack: Automated mining of silent vulnerability fixes. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 705–716. IEEE, 2021.

[Zhou *et al.*, 2022] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. SPI: automated identification of security patches via commits. *ACM Trans. Softw. Eng. Methodol.*, 31(1):13:1–13:27, 2022.