Multi-Granularity Detector for Vulnerability Fixes

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 D. Le, and David Lo

Abstract—With the increasing reliance on Open Source Software, users are exposed to third-party library vulnerabilities. Software Composition Analysis (SCA) tools have been created to alert users of such vulnerabilities. SCA requires the identification of vulnerability-fixing commits. Prior works have proposed methods that can automatically identify such vulnerability-fixing commits. However, identifying such commits is highly challenging, as only a very small minority of commits are vulnerability fixing. Moreover, code changes can be noisy and difficult to analyze. We observe that noise can occur at different levels of detail, making it challenging to detect vulnerability fixes accurately.

To address these challenges and boost the effectiveness of prior works, we propose MiDas (Multi-Granularity Detector for Vulnerability Fixes). Unique from prior works, MiDas constructs different neural networks for each level of code change granularity, corresponding to commit-level, file-level, hunk-level, and line-level, following their natural organization. It then utilizes an ensemble model that combines all base models to generate the final prediction. This design allows MiDas to better handle the noisy and highly imbalanced nature of vulnerability-fixing commit data. Additionally, to reduce the human effort required to inspect code changes, we have designed an effort-aware adjustment for MiDas's outputs based on commit length. The evaluation results demonstrate that MiDas outperforms the current state-of-the-art baseline in terms of AUC by 4.9% and 13.7% on Java and Python-based datasets, respectively. Furthermore, in terms of two effort-aware metrics, EffortCost@L and Popt@L, MiDas also outperforms the state-of-the-art baseline, achieving improvements of up to 28.2% and 15.9% on Java, and 60% and 51.4% on Python, respectively.

Index Terms—Vulnerability-fixing commit identification, Deep Learning, Ensemble Learning, Software Security, Software Component Analysis

1 Introduction

Nowadays, software projects are more and more reliant on third-party libraries, therefore exposed to these libraries' vulnerabilities. As an example, a vast number of applications and cloud services that use Log4J, including Steam, Apple iCloud, and Minecraft, are affected by the Log4Shell vulnerability [1], [2]. Log4Shell targets Log4J, one of the most popular Java libraries for logging messages and errors in the Java ecosystem. By logging a URI that points to a potentially untrusted Java class, attackers trick the client applications into executing malicious code.

To avoid similar attacks, there has been increasing attention to addressing the growing problem of vulnerabilities propagated through libraries in a software ecosystem [3], [4], [5]. As developers are slow in updating their dependencies [6], [7], [8], [9], [10], [11], tools have been developed to alert users of library vulnerabilities that may affect their applications [12], [13], [14], [15]. For example, the Open Web Application Security Project (OWASP[1]) foundation developed Dependency-Check [12], a tool that alerts users of publicly disclosed vulnerabilities within a project's dependencies.

These tools, which are referred to as Software Component Analysis [16], rely on databases of publicly disclosed vulnerabilities. Unfortunately, there is often a gap between the time a vulnerability is fixed and the time that a vulnerability is publicly disclosed [14], e.g., the inclusion of the vulnerability in the National Vulnerability Database (NVD). For example, the fix for Log4Shell was pushed four days before its public disclosure. This gap of time creates a window of opportunity for attacker to develop an exploit before the vulnerability is even known. If a vulnerability is unknown, tools cannot be developed to detect it. To address this problem, previous studies [17], [18], [19], [20] have propose tool to automatically detect security-relevant changes (i.e., vulnerability-fixing commits) that are not yet disclosed in open source projects.

Automatic identification of vulnerability-fixing commits has been used in many security companies such as Huawei, Veracode, Mend, and Snyk to monitored potential security issues from commits and other artifacts to provide users early warning of unpublished vulnerabilities [15], [16], [21], [22], [23]. It also can assist the security researchers in maintaining and updating the vulnerabilities database, such as National Vulnerability Database (NVD). Moreover, identifying vulnerability-fixing commits can enable applications such as hot patch generation and deployment [24] and

- *Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Chengran Yang, Zhipeng Zhao, Bowen Xu, David Lo are with the School of Computing and Information Systems, Singapore Management University, Singapore.*
  E-mail:{gtnguyen, tlecong, hjkang.2018, ratnadiraw.2020, cryang, zpzhao, bowenxu.2017, davidlo} @smu.edu.sg.
- *Jiayuan Zhou and Xin Xia are with the Software Engineering Application Technology Lab, Huawei, China.*
  E-mail: jiayuan.zhou1@huawei.com, xin.xia@acm.org
- *Ahmed E. Hassan is with School of Computing, Queen's University, Canada*
  E-mail: ahmed@cs.queensu.ca
- *Xuan-Bach D. Le is with School of Computing and Information Systems, The University of Melbourne, Australia*
  E-mail: bach.le@unimelb.edu.au
- *Bowen Xu is the corresponding author.*

1. https://owasp.org/

patch presence testing [25]. As substantial human effort is required to identify vulnerability-fixing commits manually, automated approaches to detect them are worth investigating. For example, a dataset of 1,282 vulnerability-fixing commits constructed in prior work required approximately four years to be manually curated [26]. Consequently, security companies have invested in building and deploying automated approaches to identify vulnerability-fixing commits to enhance IT supply chain security [16], [17], [20], [27].

To address this problem, previous works [16], [17], [18], [28], [29], [30] leverage related resources of commits such as commit messages or issue reports to classify commits. Unfortunately, in accordance with the good practice of coordinated vulnerability disclosure [31], [32], these resources should not mention any security-related information to fix vulnerabilities without exposing their existence before public disclosure of the vulnerability. Hence, detecting vulnerabilities and their corresponding fixes with the use of natural language resources such as commit messages or issue reports may be impractical.

Identifying vulnerability-fixing commits based on code-changes alone is an inevitable choice. However, traditional code analyses are not suitable for this task due to two main reasons: (1) most of these techniques cannot be applied to partial code, i.e. code changes in a commit [30], and (2) they require hand-crafted specifications or heuristics, which can be challenging and time-consuming to create [33]. An alternative solution is to use deep-learning-based analysis techniques, which can handle fuzzy inputs, including natural language integrated into code (e.g., meaning of variables' names [34]), and hidden patterns. These techniques have been shown to outperform traditional code analysis methods in various tasks, such as type inference [35], [36], fault localization [37], [38], [39] and program repair [40], [41], [42], [43], [44]. Inspired by this success, Zhou et al. proposed VulFixMiner [20], which utilizes CodeBERT to automatically represent code changes and extract features for identifying vulnerability-fixing commits. Their empirical evaluation demonstrated that VulFixMiner can accurately identify 49% of vulnerability-fixing commits with a minimal effort, inspecting only 5% of the total lines of codes.

Although VulFixMiner has achieved positive performance, we found that there are aspects that are worth further investigation. Commits could be tangled; a commit may contain changes related to different purposes, such as implementing new features and refactoring code [45]. In a tangled vulnerability-fixing commit, irrelevant changes may contribute to noise. The high noise may pose a challenge to a machine learning classifier. From our observations on real-world vulnerability-fixing commits, as illustrated in Section 2, noise can be presented at different levels of granularity, such as the file level, hunk level, or line level. Besides, the dataset of vulnerability-fixing commits is highly imbalanced, mainly because there are significantly fewer vulnerability-fixing commits compared to non-vulnerability fixing commits in the same project. For example, the vulnerability-fixing commits only account for 0.34% of all commits in the VulFixMiner test dataset [20]. The high data imbalance also poses a challenge to a machine learning classifier.

To address the aforementioned issues, we present MiDas (Multi-Granularity Detector for Vulnerability Fixes), an approach that constructs different base models for each level of code change granularity, corresponding to commit-level, file-level, hunk-level, and line-level, following their natural organization and then use an ensemble model combining all base models to output the final prediction. The benefit of MiDas are three-fold. Firstly, decomposing code changes into different levels of granularity allows MiDas to utilize a suitable extractor for each level, as discussed in Section 4.3.2. Secondly, ensemble learning helps to reduce errors caused by noise. According to previous research [46], individual classifiers tend to make different errors on each sample but typically agree on their correct classifications. Thus, by combining multiple classifiers, ensemble learning can reduce the impact of noise in the data by averaging out the error components. Thirdly, ensemble learning has been shown to be effective in addressing data imbalance problems, as demonstrated in previous studies [47], [48], [49].

Contribution. In this paper, we made the following contributions:

- We propose MiDas, a deep learning model, which utilizes multiple levels of granularity of code changes, along with an effort-aware adjustment to detect vulnerability-fixing commits.
- We demonstrate that our approach outperforms the current state-of-the-art approach on most of the evaluation metrics. In terms of AUC, MiDas outperforms the best baseline by 4.9% and 13.7% in Java and Python, respectively. In terms of effort-aware metrics, i.e., CostEffort and P_{opt}, MiDas improves the best baseline up to 60% and 51.4%, respectively.
- We conduct two ablation studies and find that the designs of multi-level granularities and effort-aware adjustment are effective. Specifically, compared to single-level granularity, combining multiple granularities increases the performance up to 4.9%, 8.5% and 17.9% in terms of AUC, CostEffort, and P_{opt}, respectively. Meanwhile, effort-aware adjustment boosts the performance of MiDas up to 21% and 22% in terms of CostEffort and P_{opt}, respectively.

Organization. The rest of the paper is organized as follows. Section 2 presents a motivating example that demonstrates the benefit of considering different levels of granularity for vulnerability-fixing commit detection. Section 3 introduces background of the target problems and the used techniques. Section 4 describes the overview and main components of MiDas. Section 5 compares MiDas against other baselines for the target task. Section 7 mentions the threats to validity. Section 8 introduces the related studies. Finally, Section 9 presents our conclusions and future directions.

Data Availability. To support the open science initiative, we published implementation and datasets of MiDas at

https://github.com/soarsmu/midas

2 Motivating Example

In this section, we present several motivating examples of vulnerability-fixing commits in the real applications to

ID	Code Changes	VF
1	*import java.util.logging.Logger*	✗
...	...	
6	*private static final Logger LOG = LogUtils.getL7dLogger(AttachmentDeserializer.class);*	✗
7	*private int maxHeaderLength = DEFAULT_MAX_HEADER_SIZE;*	✓
...	...	
9	*if (buffer.length() > maxHeaderLength) {*	✓
10	*LOG.fine(<LOG_CONTENT>);*	✗
11	*throw new HeaderSizeExceededException();*	✓
...	...	

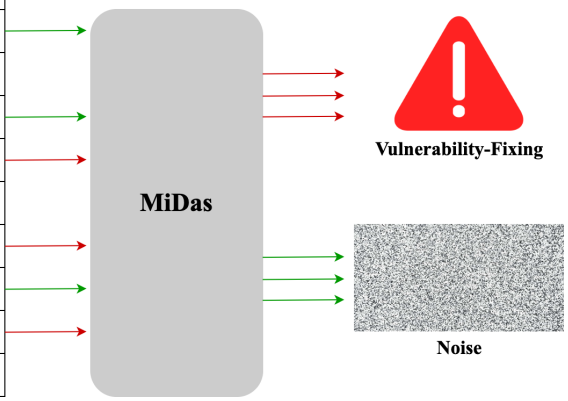

Fig. 1: A sample commit fix at line-level granularity which is for fixing CVE-2017-12624 ("VF" denotes if the code change is related to implementing the fix)

ID	Code Changes	VF
1	*FederationAuthenticationFilter.java*	✓
2	*SpringTest.java*	✗
3	*AbstractsTests.java*	✗
4	*ApplicationContext-security.xml*	✗

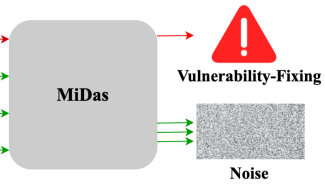

Fig. 2: A sample commit fix at file-level granularity which is for fixing CVE-2017-12631 ("VF" denotes if the code change is related to implementing the fix)

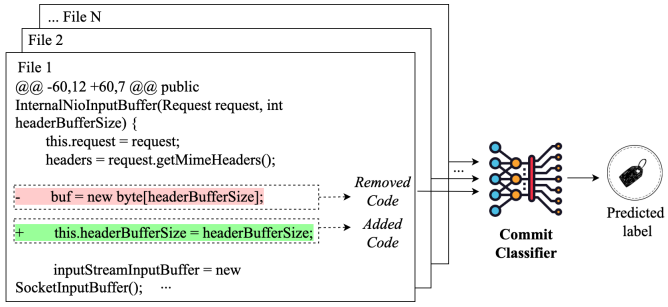

Fig. 3: Input/Output of Vulnerability-fixing Commit Classification

demonstrate the benefits of considering a commit as multi-level granularity structure data to achieve an effective classification.

Figure 1 presents a real-world commit made in Apache CXF [2] which is to fix the vulnerability CVE-2017-12624 [3], a Denial of Service (DoS) vulnerability. The root cause of the vulnerability is directly from the improper logic handling related to the constant DEFAULT_MAX_HEADER_SIZE in the source code. As we see, the code changes in this commit spread across multiple files, hunks, and lines. However, we find that the key to determining whether the commit fixes the vulnerability or not is paying attention to the code changes at line-level, which serves to fix the root cause. We observe that the remaining code changes are for other purposes, like logging and testing. For all the aforementioned reasons, we believe that either using commit-level, file-level, or hunk-level granularity is not suitable to represent code changes because applying embedding models at these levels would possibly return noisy features. Indeed, the state-of-the-art model [20], which represents code changes at file-level granularity, failed to classify this commit as a vulnerability-fixing commit.

Figure 2 shows another example commit in a real application [4]. The commit is to fix the vulnerability CVE-2017-12631 [5], which is related to Cross Style Request Forgery (CSRF). The commit contains four file changes, where only one of them is dedicated to implementing the fix, and the remaining two files are for testing. In other words, the commit is *tangled*, and this phenomenon has been proved to be common [45]. Prior works [17], [18] process all the code changes within a commit without recognizing their source files. In such a way, the code changes in the test files considered as noises in this example will be mixed with the code changes for vulnerability fixing. Thus, we find that considering the code changes of a commit at file-level granularity can help separate the code changes for vulnerability fixing from other purposes. As a result, it could further boost the performance for our target task, i.e., vulnerability-fixing commit classification.

From the above examples, they motivate us to consider features from multiple levels of granularity for the vulnerability-fixing commit classification.

3 Background

In this section, we first present the formal definition of the problem. And then, we introduce the essential background of the different types of neural networks leveraged in our approach.

2. https://github.com/apache/cxf/commit/8bd915bfd7735c248ad660059c6b6ad26cdbcdf6
3. https://nvd.nist.gov/vuln/detail/CVE-2017-12624
4. https://github.com/apache/cxf-fediz/commit/48dd9b68d67c6b729376c1ce8886f52a57df6c4
5. https://nvd.nist.gov/vuln/detail/CVE-2017-12631

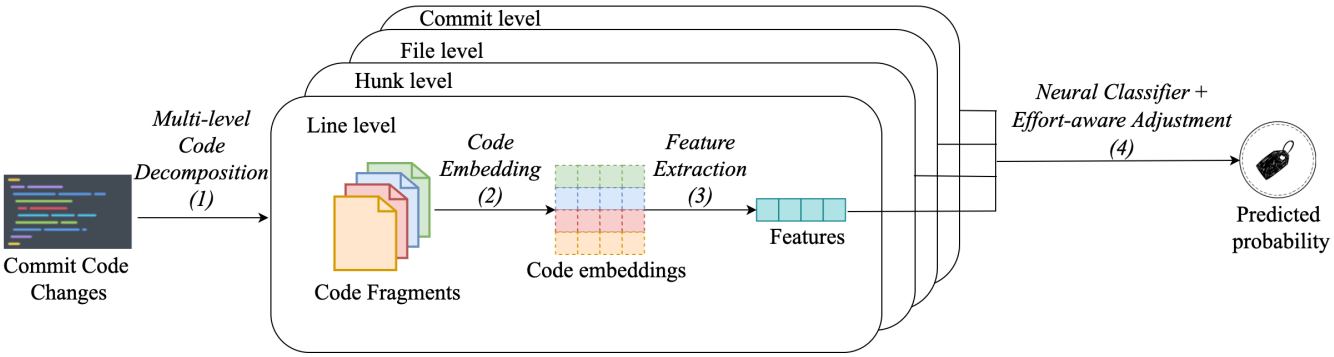

Fig. 4: Overview of MiDas

3.1 Vulnerability-fixing Commit Classification

Following many previous works, we formulate the vulnerability-fixing commit classification task as a binary classification problem. Formally, the input and output of the problem are described as follow (see Figure 3):

Input: a commit. In this task, we only consider the code changes of a commit as our input and ignore other information, e.g., commit message, by following the prior work [20]. The code changes may spread across multiple files, where code changes on the single file could consist of one or multiple hunks (i.e., groups of differing lines). Each hunk is in the form of a group of added and removed lines of code.

Output: whether the commit is for vulnerability-fixing or not. Many existing approaches derive the output by producing a probability from 0 to 1 as the likelihood that the commit is for vulnerability-fixing. The higher the probability, the more likely the commit is a vulnerability-fixing commit.

3.2 CodeBERT

CodeBERT [50] is a bimodal pre-trained model for programming language (PL) and natural language (NL) [51]. It is trained on a large-scale dataset CodeSearchNet [52] written in six programming languages, Python, Java, JavaScript, PHP, Ruby, and Go, respectively. The dataset consists of over 2.1M bimodal datapoints, which refers to pairs of NL-PL, and 6.1M unimodal datapoints, which refers to only PL.

CodeBERT considers two tasks at the pre-training stage: masked language modeling (MLM) and Replaced Token Detection (RTD). Briefly, given an input sentence where some tokens are masked out, the MLM task predicts the original tokens for those masked tokens. For the RTD task, it aims to identify which tokens are replaced from the given input. The bimodal datapoints are used for both tasks, whereas the RTD task further uses unimodal datapoints to train the model. Hence, CodeBERT is able to handle both modalities of data. The model has been proven practical in various SE-related downstream tasks, such as natural language code search [53], code document generation [53], [54], program analysis [36], [55] and program repair [56], [57], [58].

3.3 Deep Neural Networks

3.3.1 Convolutional Neural Network (CNN)

CNN [59] is a type of neural network for extracting high-level features from input data. To achieve this, a CNN model first employs convolutional layers to generate the connectivity of local input features via kernels, which are $K \times K$ weight filters. Particularly, an input and its adjacent features are multiplied with a linear filter and then summed before being added a bias term and passed through an activation function such as ReLU [60] or Sigmoid [61]. In this way, convolutional layers can capture the local correlation of the inputs. Moreover, to empower convolutional layers, CNN uses a pooling mechanism, which partitions the output of convolutional layers into several non-overlapping regions and outputs the max, min, or average of each region. The mechanism enables CNN to reduce the feature dimensions as well as keep important features. CNN has been proven its effectiveness in many SE tasks, like software question and answering posts representation [62], fault localzation [38], code generation [63], or just-in-time defect prediction [64]

3.3.2 Long Short-term Memory (LSTM)

LSTM [65] is a special kind of Recurrent Neural Networks (RNNs) capable of handling long-term dependencies in sequential data. A standard LSTM unit comprises a forget gate, an input gate, an output gate, and a memory cell. The forget gate decides information from memory that is forgotten, the input gate selects new information to update the memory, and the output gate controls the extent to the information in the memory to update the hidden state of the LSTM unit. In this way, LSTMs regulate the information that should be kept or discarded while traveling through the data sequence to avoid the problem of long-term dependencies. In this paper, to enhance the learning capability of the model, we employ an extension of LSTM, i.e., Bidirectional LSTM [66], which enables additional training via traversing the input data twice: left-to-right and right-to-left.

4 Approach

Figure 4 illustrates the overall architecture of our proposed approach for detecting vulnerability-fixing commits, namely MiDas. MiDas takes a commit as its input, then outputs the probability indicating that a commit is for vulnerability-fixing or not. More specifically, MiDas consists of five steps:

- **Multi-level code decomposition** extracts information from a commit at different levels of granularity, i.g, lines, hunks, files or a whole commit,
- **Code Embedding** encodes the extracted information at different levels of granularities into numerical

vectors by using a pre-trained model as inputs to the deep learning models in the feature extraction layers.
- **Feature Extraction** extracts features of commit codes at each level of granularity. The features are then concatenated to form the final representation of the input commit.
- **Neural Classifier** learns the mapping from the final representation of the input commit to the corresponding output vector in the training stage and then infers the likelihood that the commit is for vulnerability fixing.
- **Effort-aware Adjustment** adjusts output probability of neural classifier to guarantee our system performance with limited human efforts.

In the rest of the section, we introduce each step with more details.

4.1 Multi-level Code Decomposition

By design, a commit is in the form of code changes applied on a set of files. Each hunk shows one area where the files differ and it is in the form of a sequence of code changes applied on lines of code (LOC). Considering the structure of commits, our approach extracts information from a commit at different levels of granularities in this step. To achieve this, it decomposes a commit into code fragments corresponding to four levels of granularity based on the natural organization of a commit, i.e., line, hunk, file, and commit. For example, at line-level granularity, we split code changes into lines then treat each input commit as a sequence of LOC. As a result of this step, we obtain representations of the input commit at four levels of granularity as follows:

- **Commit-level**: A input commit is considered as a single code fragment by sequentially concatenating code changes of the whole commit.
- **File-level**: A input commit C is considered as a set of code fragment, $C = \{f_1, f_2, ..., f_F\}$ where,
  - F is the number of files in the input commit
  - f_i is a code fragment created by sequentially concatenating code changes of the i^{th} file in the input commit
- **Hunk-level**: A input commit C is considered as a set of code fragment, $C = \{h_1, h_2, ..., h_H\}$ where,
  - H is the number of hunks in the input commit
  - h_i is a code fragment created by sequentially concatenating code changes of the i^{th} hunk in the input commit
- **Line-level**: A input commit C is considered as a set of code fragment, $C = \{l_1, l_2, ..., l_F\}$ where,
  - F is the number of files in the input commit
  - l_i is a code fragment of the i^{th} line in the input commit

Figure 5 shows the structure of a commit. *Commit 1* involved changes in two files *File 1* and *File 2*. Following that, *Hunk 1* and *Hunk 2* in *File 1*, and *Hunk 3* and *Hunk 4* in *File 2* were modified, respectively. In every hunk, each of them contains 2 modified lines, from *Line 1* to *Line 8*. As

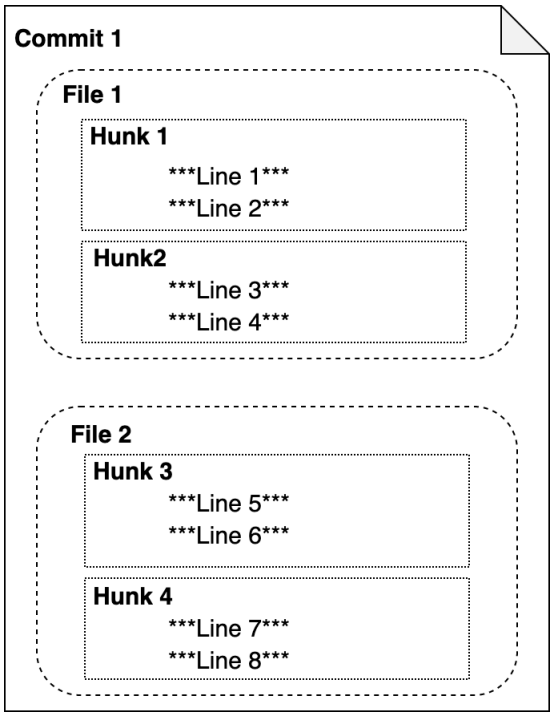

Fig. 5: An example for extracted code fragments for different granularity after multi-level code decomposition

the result of multi-level code decomposition, we obtain code fragments belong to different granularity as following:

- **Commit-level**: $\{Commit\ 1\}$
- **File-level**: $\{File\ 1, File\ 2\}$
- **Hunk-level**: $\{Hunk\ 1, Hunk\ 2, Hunk\ 3, Hunk\ 4\}$
- **Line-level**: $\{Line\ 1, Line\ 2, Line\ 3, Line\ 4, Line\ 5, Line\ 6, Line\ 7, Line\ 8\}$

4.2 Code Embedding

In this step, MiDas automatically represents code fragments as high-dimensional vectors. However, it faces a challenge in identifying vulnerability-fixing commits, which involves learning code representations automatically from a relatively small-scale dataset comprising less than 1,000 vulnerability-fixing commits. To overcome this challenge, MiDas leverages CodeBERT [50], which was pre-trained on a large-scale dataset and has shown good performance when fine-tuned with small datasets [56], [57], [67]. Specifically, MiDas first fine-tunes CodeBERT at each granularity level to capture the specific characteristics of code changes at each level, and then uses the fine-tuned models as code embedding models for representing code fragments.

By default, CodeBERT takes two segments as its input: one is for the data in natural language (NL), the other is in program language (PL). And its input is in the form of:

$$[CLS]\langle NL\rangle[SEP]\langle PL\rangle[EOS] \quad (1)$$

where [*CLS*], [*SEP*], and [*EOS*] are regarded as the special tokens in CodeBERT. Specifically, the [*CLS*] token defines the start of a CodeBERT sequence, followed up by natural language text. The [*SEP*] token is used to separate natural language text and program language source code. The [*EOS*]

token is put at the end of a CodeBERT sequence. For BERT-based models, e.g., CodeBERT, the network learns to generate meaningful embedding at the position of the [CLS] token during the training.

Recall that CodeBERT is pre-trained for two different modalities of data, which are *bimodal data* (i.e., pairs of natural language and source code) and *unimodal data* (i.e., source code). Hence, in our cases, we observe that code changes in an input commit, particularly, added code and removed code, can be considered in two different ways, considering the presence of source code context; we name them *context-dependent* and *context-free* representation.

Context-dependent representation. In this representation, we consider *removed code* and *added code* within a code fragment as a pair of data.

This method aims to learn a joint representation of both the code added and removed in a commit. This representation contextualizes the added code with the removed code and vice versa.

More formally, a code fragment will be represented in input format of CodeBERT as follows:

$$[CLS]\langle \textit{rem-code}\rangle[SEP]\langle \textit{add-code}\rangle[EOS] \quad (2)$$

Then, we forward this representation to CodeBERT model and we take the output at [*CLS*] token as *initial embedding* of the code fragment.

Context-free representation. In this representation, we consider *removed code* and *added code* within a code fragment as two different unimodal datapoints. This representation treats removed code and added code separately without considering their counterparts.

More formally, a code fragment will be represented in input format of CodeBERT as follows:

$$[CLS]\langle \textit{empty}\rangle[SEP]\langle \textit{add-code}\rangle[EOS] \quad (3)$$

$$[CLS]\langle \textit{empty}\rangle[SEP]\langle \textit{rem-code}\rangle[EOS] \quad (4)$$

Then, we forward these representations to CodeBERT model. In this case, we obtain two initial embeddings, one of *added code* and one of *removed code*, for the code fragment. Note that, CodeBERT can only take maximum 512 tokens. Hence, in case the input exceeds the limit, we truncate it by only consider the first 512 tokens.

By combining four levels of granularity (as discussed in Section 4.1) and two different modalities of code fragments, we obtain seven settings of commit embedding as illustrated in Table 1. Note that we leave the combination of line-level granularity and context-dependent representation for future work due to the fact that the combination requires an alignment between lines in *removed code* and *added code*, which are not available in the context of code changes [6].

4.3 Feature Extraction

We observed that the characteristics of four levels of granularity differ. Thus, to effectively extract features from each of them, we utilize different models accordingly. Overall, the feature extraction for each level of granularity follows a common structure consisting of a feature extractor followed by a feature fusion layer. Note that each base model has one feature extractor and one feature fusion layer, where the feature extractor's design is customized for each granularity, and the feature fusion layer is shared by different granularity. We present these steps in detail below.

6. The existing tool of code alignment (a.k.a differencing) such as GumTree [68], however, the accuracy of such tool is not perfect [69]

TABLE 1: Commit embedding settings

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

4.3.1 Feature Extractor

We leverage four deep learning models as feature extractors for different levels of granularity. For a commit, each feature extractor takes embedding vectors corresponding to the specific granularity as input and returns a feature vector as output. We present the detailed architecture of these models as follows.

Line-level: Since lines between code changes are read sequentially, we leverage a Recurrent Neural Network, a standard model for processing sequential data, to extract features at the line-level granularity. Particularly, we treat code changes as a sequence of lines, where each line is represented by an embedding vector as described in Section 4.2, denoted as $[l_1, l_2, ..., l_{T_{lines}}]$. And then, we employ Bi-directional LSTM (BiLSTM) model as our feature extractor. In our case, the bi-directional LSTM uses a forward LSTM that reads the commit from l_1 to $l_{T_{line}}$ and a backward LSTM that reads the commit from $l_{T_{line}}$ to l_1. We obtain final output of LSTM as the features of the commit.

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

Hunk-level: Different from lines, the hunks within a commit do not carry sequential relationship. However, there are still dependencies between hunks that are close, for example, hunks that are in the same file. The dependencies can be shared variables, constants or function calls. Hence, we use a Convolutional Neural Network, which has demonstrated its ability to capture local dependencies in many tasks, e.g., sentences modeling [70] or face recognition [71]. Specifically, given a set of embedding vectors of hunk-level code fragments decomposed from a commit, denoted as $[h_1, h_2, ..., h_H]$, we first employ convolution layers aggregate information from neighboring hunks. More formally, the features of $i-th$ hunk-level embedding vector is represented by aggregating information from neighboring embedding vectors as,

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

where w is a kernel size of the convolution layer $Conv$. Then, we employ a max-pooling layer to extract the most

important features from the input embedding vectors and obtain the final output as the features,

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

File-level: At the file level, we aim to capture high-level relationships between all code in the commit. Therefore, we use a Fully Connected Neural Network (FCN) to capture the relationships of all files in a commit simultaneously. Specifically, give a set of embedding vectors of file-level code fragments decomposed from a commit as $[f_1, f_2, ..., f_F]$, we first represent the commit by concatenating features of all vectors from f_1 to f_F. As a result, we obtain a $n \times F$ dimensional vector as the representation of the commit, n is the vector dimension of each file (i.e., output size of codeBERT). Note that a fully connected layer often requires a fixed size of input features. Hence, to deal with the problem, we set F as a predefined parameter. For each commit, if its number of files is smaller than F, we add some blank files so that all commits have the same number of files. Otherwise, we truncate it to only its first F files. After obtaining a fixed size input vector, we employ a fully connected layer to obtain the output features, as follows:

$$f_{file} = FCN(f_1 \oplus f_2 \oplus \cdots \oplus f_F) \quad (8)$$

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

Commit-level: Similar to file-level feature extraction, we also use a fully connected layer. Specifically, given x is the commit-level embedding vector of a given commit produced by CodeBERT. We employ a fully connected layer to obtain the output features, as follows:

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

where FCN is a fully connected layer with input size and output size of n with n is the size of x, i.e., output size of codeBERT.

4.3.2 Feature Fusion

Based on the extracted feature vectors, we further construct a set of fully-connected layers as our feature fusion. Note that, due to the different types of code embedding discussed in Section 4.1, we have two different feature fusion, i.e., bimodal and unimodal fusion corresponding two representation (i.e., *context-dependant* and *context-free* representation) methods as follows:

- *Bimodal fusion*: As mentioned in the previous section, we only obtain one feature vector for context-dependant representation. Thus, we directly feed it to a linear layer to fuse the features.
- *Unimodal fusion*: In this case, we have two feature vectors, one for *added code* and one for *removed code*. Hence, we first concatenate them into one vector then feed the vector into a linear layer to fuse the features.

4.4 Classifier and Effort-aware Adjustment

4.4.1 Neural Classifier

Given extracted features of a commit from our extractors (as discussed in Section 4.3.2), we use a neural network classifier to indicate whether the commit is for vulnerability-fixing or not. To achieve that, we first concatenate features of a commit, which is extracted at multiple granularities, then forward it into two fully connected layers to estimate a probability that the given commit is for fixing a vulnerability.

4.4.2 Effort-aware Adjustment

To increase the number of detected vulnerability-fixing commit under a limited inspection cost, i.e., the inspected line of codes (LOC), we propose an *effort-aware adjustment* as a post-processing step. The adjustment aims to adjust the output of our vulnerability-fixing classifier based on the length of commit to prioritize the shorter vulnerability-fixing commits over the longer ones. Specifically, our effort-aware adjustment is defined as follows:

$$\mathcal{P}(c) = prob_c \times f(loc_c) \quad (10)$$

Where P(c) is the adjustment applied to the probability predicted by the neural classifier, denoted as $prob_c$, for a given commit c. We want P(c) to be proportional to the number of LOC of c, loc_c. The greater the number of LOC, the greater the adjustment. Therefore, we denote $f(loc\_c)$ as a function of loc_c that would satisfy this property. Nevertheless, $f(loc_c)$ should be carefully designed so that the adjustment does not dominate the probability predicted by the neural classifier. Hence, we choose the logarithm function as our f function. More formally,

$$f(loc_c) = log_a(loc_c) \quad (11)$$

In Equation 11, a is the maximum number of LOCs of the vulnerability-fixing commits in the training dataset. As loc_c is greater or equal to 1 and less than a, $log_a(loc_c)$ is bounded from 0 to 1 for any commit in the training dataset. As a result, we have modified Equation 10 as follows:

$$\mathcal{P}(c) = prob_c \times log_a(loc_c) \quad (12)$$

Based on proposed effort-aware adjustments, we adjust the output probability of neural classier to obtain the final score of each commit as follows:

$$\mathcal{S}(c) = prob_c - \mathcal{P}(c) \quad (13)$$

Where c is a given commit, $prob_c$ is the output probability of the neural classifier, and P(c) is the calculated value of effort-aware adjustment for c. However, in the real world, there may be vulnerability-fixing commits with lengths greater than a. It would lead to a negative *S(c)* in Equation 13. As we favor shorter commits for inspection, these large commits will be poorly ranked; thus, we ignore these outliers. To preserve the correctness of our evaluation, we limit *S(c)* to 0. Hence, Equation 13 can be written as follows:

$$\mathcal{S}(c) = \max(prob_c - \mathcal{P}(c), 0) \quad (14)$$

To summarize, our effort-aware adjustment function will modify the predicted probabilities of all commits in the test dataset. This modification affects probability-based evaluation metrics, including AUC, CostEffort, and P_{opt}, which we will discuss further in Section 5.

4.5 Training

In this section, we discuss about the process of training MiDas, including training strategy and optimization.

4.5.1 Training Strategy

As mention in Section 4, MiDas employs multiple feature extractors, corresponding to different commit embedding settings (refer to Table 1). Technically, fully training MiDas is too expensive because it would require extensive resources of hardware and time. Therefore, we split the training process of MiDas into two phases, namely Base Model Training and Ensemble Training, respectively. In Base Model Training, we independently train each base model which corresponds to each commit embedding setting. Next, in Ensemble Training, we use a neural classifier to combine output features from these base models to initially obtain the predictions from MiDas .

Base Model Training The target of this phase is to train base models, in which each model consists of a CodeBERT and a feature extractor, to classify commits with respect to the corresponding embedding setting. Ideally, we want to train each base model in one fold. However, because using CodeBERT is resource-expensive, one-fold training is only applicable for base models in which the number of code fragments for one commit is small, i.e., commit-level and file-level base models. In other base models (i.e., line-level and hunk-level base models), we split this training phase into two steps. The first step is to fine-tune CodeBERT to predict if a code fragment is for vulnerability-fixing or not. As our dataset contains only the ground-truth label for the entire commit, to finetune CodeBERT, we heuristically consider that a code fragment is vulnerability-related if it belongs to a vulnerability-fixing commit. After finetuned, we freeze all CodeBERT's parameters and use embedding extracted by CodeBERT to train the corresponding feature extractor.

Ensemble Training In this phase, we freeze all parameters of base models, which are pre-trained in the previous phase, and only train the neural classifier.

4.5.2 Optimization

As MiDas is a vulnerability-fixing commits detector, which solves the problem belonging to binary classification, our training objective is to minimize the Cross-Entropy for the model on the entire training dataset. To update the weights of our neural networks, we use Adam optimizer [72], which is broadly used in many fields of deep learning. The learning rate is set to 1e-5 following CodeBERT [50].

For base model training, CodeBERT of each base model is fine-tuned for one epoch. After that, each base model is trained on training set. The process stops training if the value of the Cross-Entropy loss on the validation set has not been updated in the last five epochs. All base models are trained for a maximum of 60 epochs. For ensemble training, the neural classifier is trained with a learning rate of 1e-5 and 20 epochs.

4.6 Application

In an industrial setting, vulnerability-fixing commits detected through machine learning undergo a manual assessment by human experts [16], [17]. Our proposed approach MiDas supports the same setting, aiding security experts/researchers in monitoring commits. Given a set of commits as inputs, MiDas outputs a ranked list of possible vulnerability-fixing commits. Previous studies have suggested that security experts can leverage commits that address potential vulnerabilities to enhance IT supply chain security within the industry [16], [17], [20], [26]. For instance, Zhou and Sharma's approach [16] was utilized to identify vulnerability-fixing commits for developing Software Composition Analysis (SCA) database in Veracode. Similarly, Sabetta et al. [17], [26] extended this work at SAP, creating the SCA database for their vulnerability assessment tool, Eclipse Steady[7]. Additionally, SAP developed Prospector[8], which utilizes a vulnerability description in natural language as input to produce a ranked list of commits in decreasing order of relevance, thereby reducing the effort required to identify security fixes for known vulnerabilities in open-source software repositories. Zhou et al. [20] further extended this research to develop VulFixMiner, a vulnerability-fixing commit identification model for Huawei, which is proven capable of detecting unreported vulnerability-fixing commits as confirmed by security experts. In this paper, we demonstrate that our solution, MiDas, significantly outperforms the state-of-the-art VulFixMiner across multiple programming languages.

5 Evaluation

Our experiments are driven by the following research questions (RQs):

RQ1. How effective is MiDas compared to the baselines? To answer this RQ, we compare MiDas with VulFixMiner [20], the current state-of-the-art approach, which is also designed for vulnerability-fixing commit classification. We also utilized LApredict [73] and DeepJIT [64], which are the state-of-the-art approaches for buggy commit detection (e.g., JIT defect prediction). Furthermore, we investigate the technical differences between MiDas and the state-of-the-art baseline. We analyze the components of MiDas and compare the performance of different versions of MiDas with the state-of-the-art baseline.

RQ2. How does the effort-aware objective function affect the performance of MiDas? This RQ aims to investigate the contribution of our effort-aware objective function to MiDas. We answer the question by comparing the performance of MiDas in two versions, with and without the effort-aware objective function, respectively.

RQ3. How do different levels of granularity affect the performance of MiDas? The goal of this RQ is to investigate the influence of different levels of granularity on the performance of MiDas. We answer this RQ by continuously combining base models corresponding to each level of granularity and evaluating their performance on the considered evaluation metrics.

7. https://eclipse.github.io/steady/
8. https://github.com/SAP/project-kb/tree/commit-in-adv/prospector

TABLE 2: Statistics of Zhou et al. [20] dataset

Training Set									
Lang	**V.F.**				**N.V.F.**				#Projects
	#Commit	#File	#Hunk	#Line	#Commit	#File	#Hunk	#Line	
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 Set									
Lang	**V.F.**				**N.V.F.**				#Projects
	#Commit	#File	#Hunk	#Line	#Commit	#File	#Hunk	#Line	
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
Testing Set									
Lang	**V.F.**				**N.V.F.**				#Projects
	#Commit	#File	#Hunk	#Line	#Commit	#File	#Hunk	#Line	
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

V.F.: Vulnerability-fixing Commits, N.V.F.: Non-vulnerability-fixing Commits.

RQ4. Can MiDas detect vulnerability-fixing commits that involve different types of changes? To answer this question, we evaluate MiDas on commits containing 5 or more hunks. And then, we evaluate the performance of MiDas in comparison with the state-of-the-art baseline on the sub-datasets.

5.1 Dataset

To facilitate comparison, we evaluate MiDas on the dataset proposed by VulFixMiner [20] and follow exactly their dataset configuration. The dataset contains both vulnerability-fixing and non-vulnerability-fixing commits extracted from 150 Java projects and 106 Python projects. The vulnerability-fixing commits were collected from two sources. The first source is a manually curated Java vulnerability-fixing commit dataset, namely the SAP dataset [26]. The SAP dataset contains 1,055 vulnerability-fixing commits, spanning 183 Java OSS projects. These projects were identified based on data analysis at SAP while operating their vulnerability assessment tool called Vulas. The corresponding vulnerability-fixing commits were then manually collected over a period of four years by monitoring the disclosure of security advisories, not only from NVD, but also from projects-specific web pages. The dataset is verified by SAP researchers based on several resources such as code changes, commit messages, and reference issues.

The second source is all CVEs related to Java and Python disclosed by January 26, 2021. From the CVEs, Zhou et al. [20] collected 199 commits, 227 issues, 155 pull requests in Java, 288 commits, 244 issues, and 353 pull requests in Python. Then, the commits referenced in the pull requests and issues are extracted. Finally, all commits are merged into a single dataset after removing duplicate commits. For non-vulnerability-fixing commits, commits are sampled from the projects containing vulnerability-fixing commits up until February 26, 2021.

Until this point, the Java dataset contains 1,436 vulnerability-fixing commits and 839,682 non-vulnerability-fixing commits. Meanwhile, the Python dataset contains 885 vulnerability-fixing commits and 722,291 non-vulnerability-fixing commits. Afterward, Zhou et al. [20] further filtered the dataset by removing large commits that are less likely to fix vulnerabilities. The removal resulted in 474,555 non-vulnerability-fixing commits, and 1,353 vulnerability-fixing commits from 150 projects for Java. For Python, the corresponding values are 357,696 non-vulnerability-fixing commits and 751 vulnerability-fixing commits from 106 projects. Finally, Zhou et al. [20] enhance the dataset by labeling more commits that are relevant to vulnerability fixes, more specifically, commits which message contains vulnerability-related keywords (i.e., "vuln", "CVE", and "NVD"). To ensure the pattern is well-designed, Zhou et al. [20] randomly sampled a subset of extracted commits by it and manually verified them. As a result, in the Java dataset, they relabel 420 non-vulnerability-fixing commits across 123 projects. In the Python dataset, they relabel 501 non-vulnerability-fixing commits across 98 projects.

The dataset follows the standard manner that it is split into three parts without overlap of projects, training set, validation set, and testing set. Recall the dataset configuration, the dataset is split project-wise, using an 80%/20% split and consider the 20% split as testing dataset. Then, the remaining 80% is further split with the ratio 90%/10%, consider using 90% for training dataset and 10% for testing dataset. Note that the training and validation dataset are randomly under-sampled to reduce the imbalanced nature. The details of the dataset distribution are shown in Table 2.

5.2 Evaluation Metrics

To facilitate a fair comparison, we use the same evaluation metrics by following the prior work [20], they are AUC and two effort-aware metrics (i.e., CostEffort@L and P_{opt}@L).

AUC (Area Under the Curve): is the area under the Receiver Operating Characteristic (ROC) Curve [74]. It is a threshold-independent measure, which illustrates the discriminant ability of proposed techniques for binary classification problem [75]. AUC represents the probability that a randomly chosen negative example (i.e., non-vulnerability-fixing commit) will be ranked higher than a randomly chosen positive example (i.e.,vulnerability-fixing commit). More formally, AUC score is calculated as follow:

$$AUC = \frac{S_0 - n_0(n_0+1)/2}{n_0 n_1} \quad (15)$$

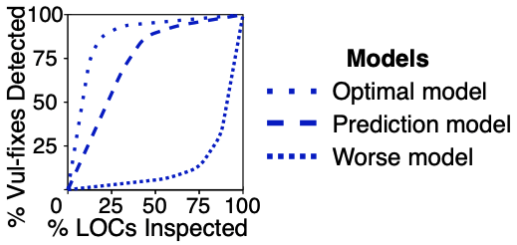

Fig. 6: An example from Zhou et al. [20] showing the relationship between the percentage of vulnerabilities fixes detected and the amount of inspection cost (i.e., % LOC) for different models

where n_0 and n_1 are the numbers of vulnerability-fixing and non-vulnerability-fixing commits, respectively, and $S_0 = \Sigma r_i$, where r_i is the rank of the i^{th} vulnerability-fixing commit in the descending list of output probability produced by each model.

CostEffort@L: The goal of a vulnerability-fixing commit detector is to rank vulnerability-fixing commits higher than the non-vulnerability fixing ones, so that, developers are capable of inspecting the code changes (i.e., the number of inspected lines of code) with a specific amount of effort. Given the commits, which are ordered by predicted probabilities obtained from the model, CostEffort@L counts the number of detected vulnerability-fixing commits, starting from commit with high to low predicted probabilities until the number of lines of code changes reaches L% of total LOC. The value of CostEffort@L represents the effectiveness of the approach under the predefined inspecting cost. The higher value of CostEffort@L, the better effectiveness of the model. In [20], only CostEffort@5% and CostEffort@20% are considered. In this work, we also calculate CostEffort@10% and CostEffort@15% to investigate how the performance differs with the increase of inspecting cost.

P_{opt}**@L**: P_{opt} is a normalized version of the cost-aware performance metric introduced by Mende and Koschke [76]. Given an Alberg diagram [77] that shows the relationship between the number of vulnerability-fixing commits (on the y-axis) and the inspection cost (on the x-axis). P_{opt}@L is computed for a given inspection cost, L, which is the percentage of total lines of code (LOCs) inspected. P_{opt} is an effort-aware performance metric used in studies on defect prediction [78], [79], [80], [81]. $P_{opt}@L$ was also used in the previous study on detecting vulnerability fixing commits [20]

Assuming we wish to assess a prediction model M, which outputs a sorted list of commits. M is compared against the optimal model, O, and the worst model, W. Using the ground-truth labels, O and W order the commits as their output. O ranks ground-truth vulnerability-fixing commits higher than non-vulnerability-fixing commits, favoring commits with fewer LOC. W ranks non-vulnerability fixing commits higher than vulnerability-fixing commits, favoring commits with a greater LOC. As such, the performance of the optimal model represents the upper bound of the performance of any prediction model, while the performance of the worst model represents the lower bound. $Curve_M$, $Curve_O$, $Curve_W$ are the curves of the prediction model M, the optimal model O, and the worst model, W, respectively (see Figure 6). For any two models, A and B, Area($Curve_A$, $Curve_B$) is the corresponding area between the curves. The points on the curves for a given L correspond to the percentage of vulnerability-fixing commits detected with L% of the total LOC inspected. For the prediction model M, $P_{opt}(M)$ is computed as:

$$P_{opt}(m) = \frac{Area(Curve_M, Curve_W)}{Area(Curve_O, Curve_W)} \quad (16)$$

A larger P_{opt} value indicates that performance between the prediction model, M, is closer to the optimal model. In our experiments, we calculate $P_{opt}@L$ with four different values of L, which are 5, 10, 15, and 20.

5.3 Baselines

We compared MiDas with the following three baselines:

VulFixMiner [20]: VulFixMiner is the current state-of-the-art baseline in vulnerability-fixing commit identification. It extracts commits at the file-level granularity and uses CodeBERT to represent code change of files. Embeddings of code changes of files are aggregated by an average function to form commit's embedding. Lastly, commit's embedding is used to train a neural classifier for prediction.

DeepJIT [64]: is a well-known deep learning approach for buggy commit identification (a.k.a defect prediction), which is relevant to our problem, i.e. vulnerability-fixing commit identification. DeepJIT takes inputs as a code change and commit message and uses deep learning models, i.e., Convolutional Neural Network, to predict whether a commit is defective or not. As our problem settings only involve code changes, we only use code change component of DeepJIT in our experiments for a fair comparison.

Other than the deep learning approaches, we compare MiDas with three simpler baselines. Sometimes, a simple model can outperform complex ones (e.g., deep learning neural networks) [73], [82]. Hence, we add the two following baselines to our evaluation:

LApredict [73]:LApredict is an approach using logistic regression with only one feature - the number of added LOCs. We selected LApredict as a simple baseline as it was shown to outperform a more complex approach [83] in identifying defective program changes. We compare MiDas with LApredict for two reasons. Firstly, LApredict is also proposed to address binary classification tasks with imbalanced data. Secondly, defects and vulnerabilities may potentially carry similar characteristics. Therefore, we want to know if LApredict can be generalized for our problem.

LOC-sensitive model: As introduced in Section 5.2, we consider CostEffort and P_{opt} as our evaluation metrics. These two metrics assess the ability to detect vulnerability-fixing commits of the model based on the certain number of inspected LOCs. Since under the same number of inspected LOCs, different models may inspect different numbers of commits, we are interested in investigating if a naive model that maximizes the number of inspected commits could

TABLE 3: Performance of MiDas and baseline models on Java and Python projects

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
	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
	MiDas	**0.85**	**0.64**	**0.77**	**0.87**	**0.91**	0.50	**0.60**	**0.67**	**0.73**
Python	**VulFixMiner**	0.73	0.32	0.40	0.48	0.56	0.24	0.30	0.35	0.39
	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
	MiDas	**0.83**	**0.47**	**0.64**	**0.74**	**0.81**	**0.33**	**0.45**	**0.53**	**0.59**

TABLE 4: Performance of MiDas with and without effort-aware adjustment on Java and Python. MiDas and MiDas$_{NoAdj}$ denote MiDas with/without adjustment, respectively

Lang	Model	AUC	CostEffort				P_{opt}			
			5%	10%	15%	20%	5%	10%	15%	20%
Java	**MiDas$_{NoAdj}$**	**0.86**	0.57	0.68	0.73	0.82	0.44	0.53	0.6	0.64
	MiDas	0.85	**0.64**	**0.77**	**0.87**	**0.91**	**0.50**	**0.60**	**0.67**	**0.73**
Python	**MiDas$_{NoAdj}$**	0.83	0.39	0.57	0.65	0.70	0.27	0.39	0.46	0.52
	MiDas	0.83	**0.47**	**0.64**	**0.74**	**0.81**	**0.33**	**0.45**	**0.53**	**0.59**

TABLE 5: Number of inspected commits at different inspection cost of LOC for Java and Python projects. MiDas and MiDas$_{NoAdj}$ denote MiDas with/without adjustment, respectively

Lang	Model	5%	10%	15%	20%
Java	**MiDas$_{NoAdj}$**	5,301	11,602	18476	25597
	MiDas	9,588	22,460	33,850	42,993
	Increment	81%	94%	83%	68%
Python	**MiDas$_{NoAdj}$**	2,677	5,689	8,896	12,390
	MiDas	4,045	8,774	14,044	18,986
	Increment	51%	54%	58%	53%

yield a good result. The intuition is that under a fixed inspection cost, the more commits that are inspected, the more vulnerability-fixing commits are detected. To do that, this naive model simply ranks commits based on the number of LOC of code changes in ascending order. Under a fixed threshold of the total number of LOC, commits with the lower number of LOCs are inspected until the threshold is met. In other words, the LOC-sensitive model assigns higher ranks for short commits than the long ones. As the other two baselines do not consider the amount of effort required, we use the LOC-sensitive model as a simple baseline that accounts for the amount of effort to make its prediction.

5.4 Experiment Results

RQ1. How effective is MiDas compared to the baselines?

To answer this question, we evaluate MiDas and baseline models on two datasets, in terms of AUC, CostEffort@k, P_{opt}@k (k equals 5, 10, 15, 20). Table 3 presents the performance results on Java and Python, respectively. Overall, MiDas outperforms all the baselines on all the evaluation metrics with one exception that VulFixMiner achieves the best performance on P_{opt}@5%.

On the Java dataset, in terms of AUC, MiDas outperforms the best baseline, i.e., DeepJIT, by 2.4% ((0.85-0.83)/0.83). Note that, except for this metric on Java, VulFixMiner is the best baseline on every metric on both Java and Python. In terms of CostEffort, the improvement achieved by MiDas over the best baseline, i.e, VulFixMiner, varies from 4.9% to 28.2% when the percentage of total LOC increases from 5% to 20% . Especially, with 20% of LOC, MiDas can identify more than 90% of the vulnerability fixes. On Popt, MiDas performs worse than VulFixMiner on Popt@5, but better by a large margin (i.e., 15.9%) on Popt@20. . Especially, with 20% of LOC, MiDas can identify more than 90% of the vulnerability fixes. On P_{opt}, MiDas performs worse than VulFixMiner on P_{opt}@5, but better by a large margin (i.e., 15.9%) on P_{opt}@20.

On the Python dataset, we find that the best performer among all the baselines is also VulFixMiner. Our model, MiDas outperforms VulFixMiner on all the metrics. In terms of AUC, MiDas leads an improvement by 13.7% ((0.83-0.73)/0.73). For the effort-related metrics, MiDas outperforms VulFixMiner by a large margin varies from 45% to 60% and from 37.5% to 51.4% on CostEffort and P_{opt}, respectively.

Besides, LApredict and LOC-based-sorting-model perform poorly on all the metrics. It suggests that these approaches are ineffective for the vulnerability-fixing commits detection problem. Since LApredict [73] only considers one feature, namely the number of LOCs, we further determine if there is a correlation between the number of LOCs and whether a commit is intended to fix a vulnerability. To do so, we follow the approach taken in prior research [84] and calculate the square of the point biserial correlation coefficient [85], denoted as *spb* . The point biserial correlation coefficient is used to measure the correlation between two variables when one of them is dichotomous, taking values of either 0 or 1. To interpret the strength of the correlation, we use the interpretation given in existing studies [84], [86], where $spb \geq 0.81$ means a very strong correlation, $0.49 \leq spb < 0.81$ indicates a strong correla-

TABLE 6: Performance of MiDas on Java and Python when continuously adding granularities

Lang	Model	AUC	CostEffort				P_{opt}			
			5%	10%	15%	20%	5%	10%	15%	20%
Java	**Commit**	0.83	0.55	0.72	0.81	0.87	0.41	0.53	0.61	0.67
	File	0.81	0.55	0.68	0.75	0.85	0.42	0.51	0.58	0.64
	Hunk	0.84	0.59	0.73	0.81	0.89	0.46	0.57	0.64	0.69
	Line	0.81	0.60	0.71	0.82	0.88	0.46	0.56	0.63	0.68
	Line + Hunk	0.84	0.62	0.74	0.84	0.90	0.49	0.59	0.66	0.71
	Line + Hunk + File	0.84	0.61	0.76	0.84	0.89	**0.50**	**0.60**	**0.67**	0.72
	MiDas (Line + Hunk + File + Commit)	**0.85**	**0.64**	**0.77**	**0.87**	**0.91**	**0.50**	**0.60**	**0.67**	**0.73**
Python	**Commit**	0.82	0.39	0.57	0.68	0.77	0.27	0.37	0.46	0.53
	File	0.80	0.43	0.56	0.65	0.74	0.27	0.38	0.46	0.52
	Hunk	0.82	0.47	0.63	0.71	0.78	0.32	0.44	0.52	0.58
	Line	0.81	0.44	0.63	0.73	0.81	0.28	0.41	0.50	0.57
	Line + Hunk	0.82	**0.48**	**0.65**	0.70	0.77	0.30	0.44	0.52	0.58
	Line + Hunk + File	0.81	0.42	0.62	**0.74**	0.79	0.28	0.41	0.50	0.57
	MiDas (Line + Hunk + File + Commit)	**0.83**	0.47	**0.64**	**0.74**	**0.81**	**0.33**	**0.45**	**0.53**	**0.59**

tion, $0.25 \leq spb < 0.49$ indicates a moderate correlation, $0.09 \leq spb < 0.25$ indicates a weak correlation, and $0.00 < spb < 0.09$ indicates very weak correlation. The calculated *spb* value is 0.00289, with a statistically significant p-value of less than 0.1, indicating a very weak correlation between the number of lines of code and whether a commit is a vulnerability fix.

From all the aforementioned results, we empirically illustrate that MiDas has higher discriminative power in identifying vulnerability-fixing commits and is able to identify more vulnerability-fixing commits under the same inspection cost compared to all other baselines.

RQ2. How does the effort-aware adjustment affect the performance of MiDas?

To answer this RQ, we compared two versions of MiDas, with and without effort-aware objective function, respectively. The experimental results for Java and Python projects are mentioned in Table 4, in which, **MiDas** denotes the performance of MiDas with the effort-aware adjustment, and **MiDas**$_{NoAdj}$ denotes the performance of MiDas without the effort-aware adjustment. Overall, although applying our effort-aware adjustment keeps AUC either remaining the same or decreases insignificantly (by 0.01), it improves the two effort-related metrics by a big margin. Specifically, on the Java dataset, the improvement in CostEffort and P_{opt} ranges from 11% to 19.1% and from 11.7% to 14%, respectively. The corresponding improvements for the Python dataset are from 12.3% to 21% and from 13.5% to 22%. The reason behind the improvement is that our effort-aware adjustment can boost the number of inspected commits without the loss of discriminative capability of MiDas, which is reflected by the stability of AUC. Indeed, as shown in Table 5, the effort-aware adjustment increases the number of inspected commits at least 68% and 51% for Java and Python projects, respectively.

Comparing the results of $MiDas_{NoAdj}$ in Table 4 with the results of VulFixMiner in Table 3, $MiDas_{NoAdj}$ outperforms the state-of-the-art baseline in terms of AUC, by 6.1% ((0.86-0.81)/0.81) on Java and 13.7% ((0.83-0.73)/0.73) on Python. This improvement comes from the difference in the neural network design of MiDas and VulFixMiner. Specifically, VulFixMiner considers only file-level granularity. Meanwhile, MiDas considers multiple granularities including commit-level, file-level, hunk-level, and line-level granularity, as described in Section 4. While, $MiDas_{NoAdj}$ underperforms on some thresholds of CostEffort@L and P_{opt}, that are CostEffort@5%, P_{opt}@5%, P_{opt}@10% on Java, by 6.6% ((0.61-0.57)/0.61), 17% ((0.53-0.44)/0.53), and 8.6% ((0.58-0.53)/0.58), respectively, by applying effort-aware adjustment, MiDas outperforms VulFixMiner on every metric (except P_{opt}@5%).

From all the aforementioned, we empirically demonstrated that the effort-aware adjustment increases the number of identified vulnerability-fixing commits under specific costs of LOC.

RQ3. How do different levels of granularity affect the performance of MiDas?

To answer this RQ, we compare the performance of multiple versions of MiDas. First, we have four versions of MiDas where in each version, MiDas contains only one granularity. Then, starting from one version, line level as an instance, we continuously integrate more levels of granularity, i.e., hunk-level, file-level, and commit-level until the complete version of MiDas is constructed. Note that effort-aware adjustment is applied for every version of MiDas.

The performance on the Java and Python dataset is shown in Table 6. Comparing four versions of MiDas that contain single granularity, we can observe that no version clearly outperforms the others across all metrics. However, the complete version of MiDas demonstrates the best overall performance. This performance improvement can be attributed to the advantages obtained from combining the different granularities. To further clarify this, we inspect the performance of MiDas while incorporating additional granularities on top of the line level. For Java, the experimental results in terms of AUC, CostEffort, P_{opt} keep increasing when a new level granularity is added continuously. It indicates that all levels of granularity contribute to the performance of MiDas. Specifically, in terms of AUC, MiDas improves 4.9% ((0.85-0.81)/0.81) compared to single level of granularity, i.e., line-level granularity. In terms of CostEffort and P_{opt}, the maximum improvements are 8.5% at CostEffort@10% and 8% at P_{opt}@5% respectively. For Python, although the experimental results are not linearly increased when each of the levels of granularity is added, the performance of MiDas still increased AUC by 2.5%

TABLE 7: Performance of MiDas and VulFixMiner on tangled Java and Python commits

Lang	Model	AUC	CostEffort				P_{opt}			
			5%	10%	15%	20%	5%	10%	15%	20%
Java	**VulFixMiner**	0.83	0.64	0.70	0.72	0.74	0.53	0.60	0.64	0.66
	MiDas	**0.89**	**0.68**	**0.80**	**0.87**	**0.90**	**0.56**	**0.65**	**0.71**	**0.76**
Python	**VulFixMiner**	0.81	0.44	0.46	0.56	0.62	0.26	0.36	0.41	0.46
	MiDas	**0.89**	**0.46**	**0.62**	**0.74**	**0.90**	**0.28**	**0.42**	**0.51**	**0.58**

TABLE 8: Regular expression used to filter security-related commits provided by Zhou et al. [16]

Rule name	Regular Expression
strong_vuln _patterns	`|(?i)(denial.of.service|\bXXE\b|remote.code.execution|\bopen.redirect|OSVDB|\bXSS\b| |\bReDoS|\bCVE\b|\bvuln\b|\bNVD\b|malicious|x-frame--options|attack|cross.site|exploit| |directory.traversal|\bRCE\b|\bdos\b|\bXSRF\b|clickjack|session.fixation|hijack| |advisory|insecure|security|\bcross--origin\b|unauthori[z|s]ed|infinite.loop)`
medium_vuln _patterns	`|(?i)(authenticat(e|ion)|bruteforce|bypass|constant.time|crack|credential| |\bDoS\b|expos(e|ing)|hack|harden|injection|lockout|overflow|password|\bPoC\b|proof.of.concept| |poison|privelage|\b(in)?secur(e|ity)|(de)?serializ|spoof|timing|traversal)`

((0.83-0.81)/0.81). In terms of effort-aware metrics, the maximum improvements are 6.8% at CostEffort@5% and 17.9% at P_{opt}@5%. Compared to the single level of granularity, i.e., line-level, MiDas either outperforms or tie on the remaining thresholds of the effort-related metrics.

Interestingly, MiDas utilizing only line-level information even can outperform VulFixMiner on the Python dataset, and demonstrates comparable performance on the Java dataset. This is due to two main reasons. Firstly, breaking down code changes into smaller, more detailed parts allows the deep learning model to consider more meaningful representations by capturing the inter-dependencies between these components, which has been shown to be effective in prior research [83]. Secondly, the gating mechanism of the LSTM enables it to selectively update and retain pertinent information, while disregarding irrelevant information. However, it is important to acknowledge that noise can arise at multiple levels, not solely at the line level as illustrated in our motivating example. Therefore, integrating information from all granularities helps MiDas to achieve best performance.

RQ4. Can MiDas detect vulnerability-fixing commits that involve different types of changes?

From the test dataset of Java and Python projects, we extract commits with a large number of disjoint code changes. Specifically, we select commits with five or more hunks. Next, we use the same evaluation metrics in the paper to compare MiDas and VulFixMiner in the subsets of Java and Python data. Table 7 present the experimental results.

Overall, MiDas outperforms the state-of-the-art baseline on all the evaluation metrics. On Java, MiDas outperforms VulFixMiner by 7.2% ((0.89-0.83)/0.83) in terms of AUC. Similarly, for Python, MiDas outperforms VulFixMiner by 9.9% ((0.89-0.81)/8.01). In terms of CostEffort@L%, MiDas improved over VulFixMiner by up to 21.6% ((0.90-0.74)/0.74) and 45.1% ((0.90-0.62)/0.62). Similarly, P_{opt} reaches the highest improvements at 20% total LOC, with 15.2% ((0.76-0.66)/0.66) and 26.1% ((0.58-0.46)/0.46) on Java and Python respectively.

Combined with the results in Table 3, our experiments indicate that MiDas has higher discriminative power on commits that have a greater number of hunks. MiDas achieves higher AUC on both Java (0.90 versus 0.85) and Python (0.89 versus 0.83). In terms of CostEffort and P_{opt}, MiDas similarly outperforms VulFixMiner at all thresholds.

6 Discussion

6.1 Can MiDas distinguish between vulnerability-fixing commits and other type of security-related commits?

As developers may make changes to secure software, not all security-related commits are vulnerability-fixing. To assess if MiDas can distinguish between vulnerability-fixing and other security-related changes, we extract a subset of the data that includes only vulnerability-fixing commits and other types of security-related commits. From both Java and Python test datasets, we extract security-related commits by using the regular expressions (see Table 8) provided by Zhou et al. [16]. We extract security-related commits from the non-vulnerability-fixing commits by matching them against the regular expressions. A total of 4,023 commits from the Java dataset and 1,455 commits from the Python dataset are extracted. Then, these commits are combined with the 300 vulnerability-fixing commits from the Java dataset, and 195 vulnerability-fixing commits from the Python dataset.

Table 9 shows the performance of MiDas and VulFixMiner when using only security-related commits for Java and Python, respectively. On both the Java and Python datasets, MiDas achieves AUC scores of 0.79 and 0.73. Following Romano et al. [87], a classifier with an AUC $\geq$ 0.7 is considered to have achieved acceptable performance. Compared to VulFixMiner, MiDas performs equally on the Java dataset with a 0.79 AUC score. On the Python dataset, MiDas improves VulFixMiner by 9% on AUC ((0.73-0.67)/0.67). Regarding CostEffort and P_{opt}, MiDas outperforms VulFixMiner on every threshold by significant margins. For Java, the improvement varies from 37% to 63% and 60% to 117.6% on CostEffort and P_{opt}, respectively. For Python, the improvement ranges from 67.5% to 122.2% and from 100% to 187.5% on CostEffort and P_{opt}, respectively. Overall, the experimental results indicate that both MiDas

TABLE 9: Performance of MiDas and VulFixMiner on dataset of security-related commits on Java and Python

Lang	Model	AUC	CostEffort				P_{opt}			
			5%	10%	15%	20%	5%	10%	15%	20%
Java	**VulFixMiner**	**0.79**	0.27	0.34	0.47	0.54	0.17	0.24	0.30	0.35
	MiDas	**0.79**	**0.44**	**0.57**	**0.66**	**0.74**	**0.37**	**0.45**	**0.51**	**0.56**
Python	**VulFixMiner**	0.67	0.25	0.27	0.36	0.40	0.16	0.22	0.27	0.29
	MiDas	**0.73**	**0.50**	**0.60**	**0.67**	**0.67**	**0.46**	**0.51**	**0.55**	**0.58**

TABLE 10: Performance of MiDas using different PCA settings on Java and Python projects

Lang	Model	AUC	CostEffort				P_{opt}			
			5%	10%	15%	20%	5%	10%	15%	20%
Java	MiDas$_{PCA_80\%}$	0.25	0.11	0.15	0.22	0.25	0.06	0.1	0.13	0.15
	MiDas$_{PCA_85\%}$	0.63	0.50	0.62	0.70	0.75	0.36	0.46	0.53	0.58
	MiDas$_{PCA_90\%}$	0.48	0.28	0.42	0.55	0.66	0.14	0.25	0.32	0.40
	MiDas$_{PCA_95\%}$	0.76	0.56	0.70	0.81	0.87	0.4	0.52	0.60	0.66
	MiDas$_{PCA_99\%}$	0.83	0.62	0.76	0.83	0.87	**0.51**	**0.60**	**0.67**	0.72
	MiDas	**0.85**	**0.64**	**0.77**	**0.87**	**0.91**	0.50	**0.60**	**0.67**	**0.73**
Python	MiDas$_{PCA_80\%}$	0.27	0.06	0.13	0.17	0.2	0.03	0.06	0.09	0.12
	MiDas$_{PCA_85\%}$	0.70	0.48	0.63	0.70	0.75	0.29	0.44	0.51	0.57
	MiDas$_{PCA_90\%}$	0.48	0.09	0.24	0.33	0.41	0.05	0.16	0.17	0.22
	MiDas$_{PCA_95\%}$	0.75	0.36	0.48	0.61	0.72	0.25	0.33	0.41	0.47
	MiDas$_{PCA_99\%}$	0.80	0.44	0.58	0.72	0.75	0.31	0.41	0.49	0.55
	MiDas	**0.83**	**0.47**	**0.64**	**0.74**	**0.81**	**0.33**	**0.45**	**0.53**	**0.59**

TABLE 11: Performance of MiDas with and without feature selection (FCBF) on Java and Python projects

Lang	Model	AUC	CostEffort				P_{opt}			
			5%	10%	15%	20%	5%	10%	15%	20%
Java	**MiDas$_{FCBF}$**	0.47	0.33	0.51	0.59	0.68	0.20	0.31	0.39	0.45
	MiDas	**0.85**	**0.64**	**0.77**	**0.87**	**0.91**	**0.50**	**0.60**	**0.67**	**0.73**
Python	**MiDas$_{FCBF}$**	0.52	0.3	0.44	0.54	0.62	0.18	0.27	0.35	0.41
	MiDas	**0.83**	**0.47**	**0.64**	**0.74**	**0.81**	**0.33**	**0.45**	**0.53**	**0.59**

and VulFixMiner can distinguish vulnerability fixes from other changes to security components.

6.2 Is there redundancy among the features extracted by MiDas?

To understand the importance of the extracted features, we compare the performance of MiDas with and without applying feature reduction, Principal Composition Analysis - PCA [88], or feature selection technique (Fast Correlation-based Feature Selection - FCBF [89])

MiDas with PCA. To study the redundancy of features, PCA has been applied in different studies, including software engineering [90], [91], [92]. Similarly, in our case, it can be used to reduce the feature space of the input vector to the neural classifier. Specifically, after obtaining the features from different granularities, we use Principal Component Analysis (PCA) to obtain the principal components and use them as inputs for the neural classifier. If the principal components obtain the same performance as the original feature vectors, it implies that some original features were redundant.

PCA computes new features called principal components, obtained from linear combinations of the original features [93]. PCA obtains these features by projecting the original features onto a lower dimensional space such that the variance of the projected data is maximized. The principal components are computed such that the first principal component will explain the most variance in the dataset, followed by the second component, and so on [93]. Hence, to assess if feature vectors extracted by different granularities are important for MiDas, we concatenate them into a vector, and we perform PCA on the combined vector before passing the principal components to the neural classifier. Following Kondo et al. [94], PCA is configured so that it explains a specific proportion of variance in the data. In our experiments, we opt to retain 80%, 85%, 90%, 95%, and 99% of the variance in the data, respectively.

Table 10 illustrates the performance of MiDas in these cases. As we can see, MiDas performs worse using the principal components. Without PCA, MiDas achieves the highest scores in every evaluation metric on both Java and Python (except P_{opt}@5% on Java, with a marginal 0.01 decrease). Thus, feature selection does not help increase the performance of MiDas.

MiDas with FCBF. Fast Correlation-based Feature Selection (FCBF) [89] is a feature selection technique, which has been shown to be effective in removing redundant features in different tasks [95], [96], [97], [98]. Unlike feature reduction techniques, which compute a new set of features, feature selection techniques such as FCBF selects the most important features to be retained, removing other features. Similar to PCA, we apply FCBF after concatenating all feature vectors from different granularities. Then the output of the FCBF is passed as the input to the neural classifier.

Table 11 illustrates the performance of MiDas with and without using FCBF on Java and Python. After applying

TABLE 12: Performance of MiDas when using a different model to extract features for code at one level of granularity for Java and Python projects

Lang	Model	AUC	CostEffort				P_{opt}			
			5%	10%	15%	20%	5%	10%	15%	20%
Java	MiDas$_{Line_LSTM}$	0.84	0.62	0.74	0.84	0.88	0.49	0.59	0.66	0.71
	MiDas$_{Line_GRU}$	**0.85**	0.62	0.75	0.84	0.89	0.49	0.59	0.66	0.71
	MiDas$_{Hunk_FCN}$	0.83	0.62	0.74	0.83	0.88	0.49	0.59	0.66	0.71
	MiDas$_{File_CNN}$	0.84	0.62	**0.77**	0.84	0.90	**0.50**	0.60	0.66	0.72
	MiDas	**0.85**	**0.64**	0.77	**0.87**	**0.91**	**0.50**	**0.60**	**0.67**	**0.73**
Python	MiDas$_{Line_LSTM}$	0.81	0.42	0.64	0.7	0.78	0.28	0.41	0.5	0.56
	MiDas$_{Line_GRU}$	0.81	0.43	0.63	0.69	0.79	0.28	0.41	0.49	0.56
	MiDas$_{Hunk_FCN}$	0.81	0.46	0.61	0.71	0.78	0.29	0.42	0.50	0.56
	MiDas$_{File_CNN}$	0.82	**0.51**	**0.66**	0.70	0.79	0.32	0.45	**0.53**	0.58
	MiDas	**0.83**	0.47	0.64	**0.74**	**0.81**	**0.33**	**0.45**	**0.53**	**0.59**

FCBF, the performance of MiDas is reduced on every evaluation metric. For example, by applying FCBF, the AUC scores drop by 45% ((0.85-0.47)/0.85) and 37% ((0.83-0.52)/0.83 on Java and Python, respectively. The results suggest that MiDas does not benefit from feature selection. As neither feature reduction nor feature selection improves the performance of MiDas, we conclude that there is a low level of redundancy among the features extracted by MiDas.

6.3 Does the choice of neural network for feature extractor affect the performance of MiDas?

As described in Section 4.3.1, MiDas uses different deep learning models to extract code features at different granularity levels. Therefore, we perform a set of experiments to observe the performance of MiDas when using different deep learning models for extracting features. In each experiment, we replace the current feature extractor model at one granularity with another design. Specifically, for line-level granularity, we replace our design BiLSTM with either LSTM or GRU. We denote the two corresponding versions of MiDas when using these two models at line level granularity as MiDas$_{Line_LSTM}$ and MiDas$_{Line_GRU}$, respectively. Similarly, for hunk-level granularity, we replace CNN with FCN, and for file-level granularity, we replace FCN with CNN. The replacements yield two other versions of MiDas, namely MiDas$_{Hunk_FCN}$ and MiDas$_{File_CNN}$. Table 12 shows the performance of MiDas for Java and Python when using different neural network models for a level of granularity.

Compared to the variants of MiDas where the feature extractor for one level of granularity uses a different model, MiDas achieves the highest AUC on both Java and Python, with higher scores of either 0.01 or 0.02. However, on the effort-aware metrics, MiDas$_{File_CNN}$ outperforms MiDas on CostEffort@5% and CostEffort@10% on the Python dataset by 8.5% and 3.1%, respectively. Overall, we see that different model designs slightly affect the performance of MiDas. Nevertheless, when using the proposed design in Section 4.5.1, MiDas achieves the highest results on most evaluation metrics. It confirms our intuition in designing the feature extractors.

6.4 How does MiDas perform in different contexts of inspection cost?

As the current effort-aware metrics uses LOC as a measure of the inspection effort, we are curious about the performance of MiDas and the state-of-the-art baseline, VulFixMiner, when using other measures of effort, e.g., the number of hunks, files, commits inspected. Specifically, similar to CostEffort@L% described in Section 5.2, we defined CostEffort_Hunk@L%, CostEffort_File@L%, CostEffort_Commit@L% which are the CostEffort calculated using the number of inspected hunks, files, commits respectively. The results are illustrated in Tables 13 14, 15. Combined with the results in Tables 3, our experimental results show that on four measures, LOC, hunk, file, and commit, MiDas either outperforms VulFixMiner or performs similarly. This validates our findings from before that MiDas leads to a reduction in effort compared to VulFixMiner.

TABLE 13: Performance of MiDas and VulFixMiner for the Java and Python projects on CostEffort@L% Hunk level

Lang	Model	CostEffort_Hunk			
		5%	10%	15%	20%
Java	**VulFixMiner**	**0.54**	**0.63**	0.66	0.69
	MiDas	**0.54**	**0.63**	**0.67**	**0.72**
Python	**VulFixMiner**	0.31	0.39	0.46	0.53
	MiDas	**0.46**	**0.63**	**0.72**	**0.79**

TABLE 14: Performance of MiDas and VulFixMiner for the Java and Python projects on CostEffort@L% File level

Lang	Model	CostEffort_File			
		5%	10%	15%	20%
Java	**VulFixMiner**	**0.58**	0.64	0.67	0.70
	MiDas	0.57	**0.67**	**0.75**	**0.81**
Python	**VulFixMiner**	0.30	0.37	0.45	0.50
	MiDas	**0.43**	**0.58**	**0.65**	**0.72**

TABLE 15: Performance of MiDas and VulFixMiner for the Java and Python projects on CostEffort@L% Commit level

Lang	Model	CostEffort_Commit			
		5%	10%	15%	20%
Java	**VulFixMiner**	**0.54**	**0.63**	0.66	0.69
	MiDas	**0.54**	**0.63**	**0.67**	**0.72**
Python	**VulFixMiner**	0.28	0.37	0.44	0.50
	MiDas	**0.41**	**0.57**	**0.63**	**0.68**

TABLE 16: Number of detected large vulnerability-fixing commits of MiDas and VulFixMiner for Java and Python projects

Lang	Model	Inspection Cost			
		5%	10%	15%	20%
Java	VulFixMiner	85	96	99	101
	MiDas	**89**	**100**	**109**	**114**
	Total No. VF commits	131			
Python	VulFixMiner	8	10	12	14
	MiDas	**11**	**14**	**15**	**15**
	Total No. VF commits	26			

TABLE 17: Number of detected small vulnerability-fixing commits of MiDas and VulFixMiner for Java and Python projects

Lang	Model	Inspection Cost			
		5%	10%	15%	20%
Java	VulFixMiner	**11**	11	11	11
	MiDas	10	**14**	**14**	**15**
	Total No. VF commits	15			
Python	VulFixMiner	9	13	15	15
	MiDas	**12**	**17**	**20**	**26**
	Total No. VF commits	30			

6.5 How does MiDas perform on large/small datapoints?

We study the effect of commit size on the performance of MiDas. In particular, we investigate the performance of MiDas on large and small commits. We consider commits that exceed the limit of CodeBERT, i.e., 512 tokens, as large commits. For small commits, we selected code changes with less than 50 tokens. Our experimental results are illustrated in Tables 16, 17.

From the tables, across the programming languages, size, and inspection cost settings, MiDas outperforms VulFixMiner on 15 out of the 16 settings. The result shows that MiDas can detect vulnerability-fixing commits even when the commits are large or small, and does so better than VulFixMiner.

6.6 In terms of effort-aware metrics, why does MiDas perform better on Java compared to Python despite the same AUC?

As shown in Table 3, while MiDas achieves similar AUC scores on the Java and Python datasets (0.85 versus 0.83), there is a considerable difference between the performance of MiDas in effort-aware metrics (i.e., CostEffort, P_{opt}). This shows that MiDas can be considered to be more effective on the Java dataset despite having the same predictive power on both datasets. We investigate further to shed more light on this result by analyzing the number of commits inspected at each effort threshold, which influences the computation of CostEffort and P_{opt}.

Table 18 provides the percentage of commits that are inspected from the Java and Python datasets for each effort threshold. At every considered threshold, the proportion of inspected commits from the Java dataset is higher than in Python. This implies that while the predictive power of MiDas is similar on both datasets, the number of commits inspected under the same effort thresholds is different. As more commits are inspected on the Java dataset, a greater proportion of vulnerability-fixing commits would be detected using the same amount of effort. Hence, MiDas achieves a higher CostEffort and P_{opt} on the Java dataset.

TABLE 18: Percentage of inspected commits by MiDas based on %LOC

Dataset	5%LOC	10%LOC	15%LOC	20%LOC
Java	10.9	25.5	38.4	48.8
Python	7.2	15.7	25.2	34

TABLE 19: Statistics of Zhou et al. [20] dataset following time-aware setting. We refer to vulnerability-fixing commits and non-vulnerability-fixing commits as V.F. and N.V.F, respectively.

	Training Set		Validation Set		Testing Set	
	#V.F.	#N.V.F.	#V.F.	#N.V.F	#V.F.	#N.V.F.
Java	979	105,158	110	3,364	270	13,150
Python	548	69,480	61	3,154	152	5,375

V.F.: Vulnerability-fixing Commits, N.V.F.: Non-vulnerability-fixing Commits.

6.7 MiDas under time-aware constraint

In the original setting (Section 5), our dataset is split in project-wise manner. In this case, the commits in test data are from different projects which are never seen in the training and validation sets. Nonetheless, the problem of identifying vulnerability-fixing commits can be viewed from a different perspective, i.e., how vulnerability-fixing commits evolve with time. Specifically, can a trained model of MiDas, which is based on historical vulnerability-fixing commits, correctly identify future ones? To answer it, we first sorted all the commits in the dataset in chronological order, following a study by Feargus et al. [99]. Additionally, to ensure that the training and validation data contained an adequate number of vulnerability-fixing commits, we incrementally collected data until we reached the point where 80% of the collected commits were vulnerability-fixing commits. As a result, we produced a dataset whose statistics are described in Table 19.

Subsequent to training MiDas and the best baseline, VulFixMiner, on the new dataset, we evaluate their performance in terms of AUC, CostEffort, and P_{opt}. The result of the evaluation is shown in Table 20. The outcome indicates that MiDas continues to outperform VulFixMiner on both Java and Python on all evaluation metrics.

6.8 Applicability of MiDas on monitoring real project

In this discussion, we aim to assess the generalizability and applicability of MiDas on monitoring a real project. To do so, we run experiments on a new dataset collected from the TensorFlow[9] framework, which is not part of the Zhou et al. [20] dataset.

We first collected vulnerability-fixing commits associated with vulnerabilities reported in the National Vulnerability Database (NVD) from 18 September 2020 to 8 January

9. https://github.com/tensorflow/tensorflow

TABLE 20: Performance of MiDas and VulFixMiner on Zhou et al. [20] dataset following time-aware setting.

Lang	Model	AUC	CostEffort				P_{opt}			
			5%	10%	15%	20%	5%	10%	15%	20%
Java	**VulFixMiner**	0.76	0.36	0.45	0.52	0.57	0.32	0.36	0.41	0.44
	MiDas	0.77	0.50	0.63	0.74	0.78	0.40	0.49	0.56	0.61
Python	**VulFixMiner**	0.73	0.26	0.39	0.44	0.47	0.19	0.27	0.31	0.35
	MiDas	0.77	0.36	0.53	0.63	0.72	0.29	0.38	0.45	0.51

TABLE 21: Performance of MiDas on TensorFlow dataset

AUC	CostEffort				P_{opt}			
	5%	10%	15%	20%	5%	10%	15%	20%
0.88	0.81	0.92	0.93	0.94	0.58	0.74	0.80	0.84

2022, and excluded all commits whose messages contained security-related keywords proposed by Zhou et al. [20]. It resulted in a total of 284 vulnerability-fixing commits.

Next, we collected all commits of TensorFlow within the same time frame and considered them as non-vulnerability-fixing commits. Note that we exclusively considered source code files written in C or Python, as they are the primary programming languages for TensorFlow. To minimize the impact of large commits, we applied an approach similar to Zhou et al. [20] by establishing two thresholds using the 95th percentile of the total modified lines of code (310) and the number of changed files (7) of vulnerability fixes. As a result, we obtained 284 vulnerability-fixing and 16,083 non-vulnerability-fixing commits. We then chronologically split the dataset into training and testing sets in an 80:20 ratio. The setting is consistent with a real-life scenario in which vulnerability-fixing classification models are trained on historical commits and deployed to predict new ones, and mitigate time and spatial bias [99], [100], in our evaluation. Finally, we obtained a total of 200 vulnerability-fixing commits and 13,155 non-vulnerability-fixing commits for the training set. For the testing set, we obtained 84 vulnerability-fixing commits and 2,928 non-vulnerability-fixing commits.

Table 21 shows the performance of MiDas on the TensorFlow dataset. It can be seen that MiDas can effectively classify vulnerability-fixing commits in the TensorFlow framework with an AUC of 0.88. The results are comparable to its performance on Java and Python datasets which are 0.85 and 0.83, respectively, as reported in the Table 3. The results indicate that MiDas can generalize over different programming languages and projects. Moreover, these experimental results also suggest that MiDas can significantly reduce human efforts in identifying vulnerability-fixing commits. For instance, MiDas can detect 81% of vulnerabilities by examining just 5% of the lines of code, and this figure increases to 94% when examining 20% of the code. The results show that MiDas is promising on reducing human efforts on monitoring vulnerability-fixing commits from a real project, i.e, TensorFlow.

Despite its effectiveness in reducing the human effort required for monitoring vulnerability-fixing commits, MiDas still necessitates the involvement of security experts to validate the presence of such commits. This process can be prone to errors and requires security experts to have a thorough understanding of the codebase to verify these commits. Consequently, this poses a challenge when using MiDas to monitor vulnerability-fixing commits in real-world scenarios. However, it is worth noting that this challenge is not unique to our tool and the identification of vulnerability-fixing commits. Many other software engineering tasks face similar challenges, such as automated program repair [101], bug detection [102] or vulnerability detection [103]. Addressing these challenges falls beyond the scope of our current paper, and we leave them as potential directions for future research and investigation.

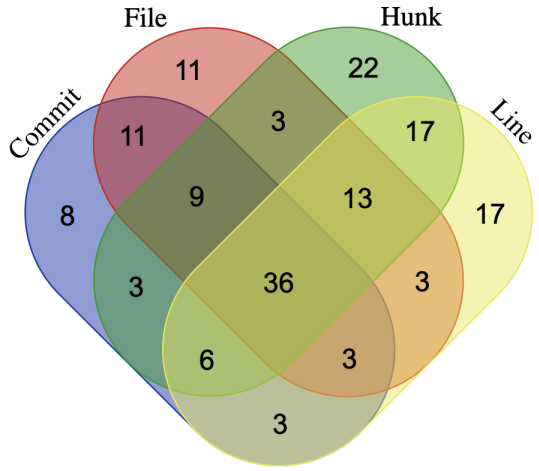

Fig. 7: Intersection of correctly detected vulnerability-fixing commits from different granularities in MiDas

6.9 Is ensemble learning needed?

Our approach MiDas employs ensemble learning. To assess the need for ensembling, we investigated whether there are unique vulnerability-fixing commits that only a specific granularity can detect. Using a threshold of 0.5, we counted the number of commits that each granularity exclusively detects, as shown in Figure 7. The figure depicts the intersection of correctly detected vulnerability-fixing commits from classifiers corresponding to different granularities. We can see that these classifiers only agree on 36 out of 165 vulnerability-fixing commits, accounting for less than 22% of the commits detected by all granularities. Furthermore, Figure 7 shows that 8, 11, 22, and 17 commits can only be detected by the Commit level, File level, Hunk level, and Line level, respectively. These results demonstrate that information from different granularities is useful for detecting different vulnerability-fixing commits. Therefore, combining information from different granularities can improve the performance of vulnerability-fixing commit classification.

TABLE 22: Mean and Standard Deviation (Stdev) in performance of MiDas and VulFixMiner for running RQ1 20 times

Lang	Model	Stat	AUC	CostEffort				P_{opt}			
				5%	10%	15%	20%	5%	10%	15%	20%
Java	VulFixMiner	Mean	0.79	0.57	0.62	0.66	0.69	0.50	0.55	0.58	0.60
		Stdev	0.018	0.020	0.023	0.021	0.022	0.014	0.014	0.017	0.018
	MiDas	Mean	0.84	0.63	0.76	0.84	0.88	0.50	0.60	0.67	0.72
		Stdev	0.003	0.007	0.010	0.009	0.007	0.004	0.004	0.006	0.005
Python	VulFixMiner	Mean	0.71	0.31	0.41	0.50	0.56	0.22	0.30	0.35	0.40
		Stdev	0.028	0.033	0.039	0.040	0.051	0.021	0.026	0.029	0.032
	MiDas	Mean	0.81	0.44	0.61	0.72	0.79	0.29	0.42	0.50	0.56
		Stdev	0.005	0.016	0.015	0.014	0.014	0.006	0.008	0.007	0.006

Indeed, MiDas achieves the best performance by using all granularities as we have shown in RQ3.

6.10 Effects of randomness on the performance of MiDas

To address the potential impact of randomness in our deep learning experiments, we ran MiDas and VulFixMiner 20 times using different seeds for each version of both models and calculated the average results and standard deviations, which are reported in Table 22. The results demonstrate that both MiDas and VulFixMiner are stable, with a standard deviation of less than 0.05.

Furthermore, to check the statistical significance of our findings, we utilized the Wilcoxon Signed-Rank Test [104], which is a non-parametric hypothesis test commonly used in previous studies [105], [106], [107], at a 95% confidence level. For each evaluation metric, we set the null hypothesis that there is no difference between the performance of MiDas and VulFixMiner, and calculated the corresponding p-value. If the p-value is less than 0.05, we reject the null hypothesis and conclude that MiDas outperforms VulFixMiner. We find that except for P_{opt}@5% on the Java dataset, all other metrics on the Java and Python datasets had p-values of less than 0.05. Therefore, we conclude that MiDas statistically significantly outperforms VulFixMiner on all metrics except Popt@5% for the Java dataset.

7 Threats to Validity

Threats to internal validity relates to the mistakes in the implementation and analysis of MiDas. To mitigate the threats, we have double-checked our source code and data. In our experiments, we used the same CodeBERT version [53] for every base model to ensure there is no difference between the used pre-trained models. Moreover, all code fragments after extracted are represented by CodeBERT in the same way as we proposed in Section 4. Our source code and data are available in our replication package [108], which future work can analyze and build on.

To minimize the threats to **construct validity**, we used the standard evaluation metrics, which have been used in numerous studies in software engineering. For a fair comparison, we also used exactly the same dataset with the same separation as the prior study [20] for our comparison with the baseline models.

To reduce the risk from threats to **external validity**, which is related to the generalizability of our findings, the dataset used in our experiments contains a large number of commits from a wide range of projects. Besides, the datasets cover two popular program languages, Java and Python. The result shows that our representations for code changes can work in both Java and Python. Additionally, we run MiDas on the TensorFlow dataset, which contains code changes in C and collected from National Vulnerability Database (NVD). The experiment reduces the risk that MiDas can only work on an artificial dataset. However, the performance of MiDas may not be generalized to other programming languages. As we use CodeBERT to represent code fragments, the quality of the representation is affected by the CodeBERT pretrained model. Specifically, CodeBERT is pre-trained on only six programming languages. It may, therefore, have limitations in representing code from other programming languages. However, according to a recent study by Chen et al. [109], language model pre-trained on high-resource programming languages (e.g., Java and Python) can be generalized even to low-resource programming languages. Depending on the downstream tasks, to achieve high performance, selecting a suitable programming language for finetuning is crucial. Therefore, we leave the analysis of MiDas on other programming languages for future work.

As our last point to reduce the risk, the proposed effort-aware objective function is applicable for different datasets, where the distribution of commits' length may not be the same as the one we used in our experiments.

8 Related Work

Many works have been proposed to identify vulnerability-fixing commits based on both commit messages and code changes, e.g., [17], [110]. Sabetta et al. [17] trained two linear Support Vector Machine models based on Bag-Of-Words representation for classifying commit message and code change, respectively. For each commit, the predictions of the two classifiers are combined using a simple voting mechanism to flag if a commit is for vulnerability-fixing or not. Zhou et al. [110] leverage LSTM and multi-layer CNN to train a commit message classifier and a code change classifier, then the results from the two classifiers are combined by using a stacking ensemble. Nguyen et al. [18] further consider the commit issue as an additional source of information for classification. Their model is an extension of the work from Sabetta et al. [17], with adding a new component of the commit issue classifier. Different from these studies, following the practice where vulnerability-related information should not be explicitly mentioned, Mi-

Das considers only code changes to identify vulnerability-fixing commits.

There also exist some works focus on other types of commit-related classification problems. For example, VCCFinder [111] utilizes an SVM-based model based on handcraft features to identify vulnerability-introducing commits. These features come from different scopes, i.e., project, author, commit, and file. DeepCVA [112] adapts multi-task learning technique to tackle the problem of characterizing vulnerability-introducing commits to provide timely information about the exploitability, impact and severity of the vulnerabilities. They use attention-based convolutional gated recurrent units to extract code change and its surrounding context within a vulnerability-introducing commit. DEPA [113] utilizes the partial-code analysis tool GRAPA [114] to analyze previous bug-fixing commits, extracting bug signatures and then employing the signatures to detect new bugs. Our work is similar in that we analyze commits to understand buggy patterns and security risks, however, our focus is vulnerability-fixing commits.

Apart from vulnerability-related commits identification, other studies have proposed methods of classifying commits based on different categorizations. DeepJIT [64] is built upon CNN to represent commit message and code changes features. Features of the two sources of information are combined by a fully connected network to predict Just-in-time defects. CC2Vec [83] is evaluated on the same task, however, only considers code changes. The core of CC2Vec is the Hierarchical Attention Network used to extract code change's feature and a set of comparison functions for capturing the difference between removed code and added code. Subsequently, LApredict [73] empirically demonstrates that a Logistic Regression model with only one feature, i.e., added-line-number, can outperform deep learning in just-in-time defect prediction.

However, the vulnerability-fixing detection task dataset is more imbalanced compared to the just-in-time defect prediction task dataset, mainly due to the limited number of existing vulnerability-fixing commits. Specifically, only 0.34% of commits in our dataset are vulnerability-fixing commits, whereas defective commits make up between 8.64% and 41.20% of all commits in the LApredict datasets [73]. Moreover, the aforementioned approaches [64], [73], [84] solely rely on code change information at a single granularity. In contrast, our study proposes an ensemble learning framework that utilizes code change information across multiple granularities to mitigate the impact of data imbalance. Ensemble learning has been proven to be effective to alleviate the data imbalance problem [47], [48], [49] by combining multiple base models and training them separately.

The most related studies to our work are VulFixMiner [20] and CoLeFunDa [27] which are also aiming to classify vulnerability-fixing commits only based on code changes. Similar to our approach, VulFixMiner and CoLeFunDa also use CodeBERT [50] to represent code changes. However, they only consider single-level (i.e., either file-level or function-level) granularity for the representation while MiDas considers multiple granularities of a code change to precisely capture fix-related information and untangle it from noise. This enables MiDas to outperform VulFixMiner in terms of discriminative ability. Moreover, MiDas uses an effort-aware adjustment function to further boost the performance of MiDas in reducing the amount of effort for inspecting the commits.

9 Conclusion and Future Work

In this paper, we propose MiDas, a multi-granularity deep learning model for vulnerability-fixing commit detection. Our findings suggest that representing commit code changes in different levels of granularity could effectively improve the performance compared to the state-of-the-art baseline. Moreover, we take the effort-aware evaluation metrics into consideration to evaluate approaches in the real-world scenario. According to the result, the proposed effort-aware adjustment function has demonstrated its effectiveness of reducing the inspection cost of developers in detecting vulnerability-fixing commits.

The current version of MiDas leverages only source code information in code changes to detect vulnerability-fixing commits. In the future, we will explore other sources of information such as code comments, project-related data, etc. to further improve our model. Besides, we plan to investigate the impact of different kinds of effort-aware adjustment functions on overall performance of MiDas. Moreover, we plan to enhance the representation of code changes by incorporating additional context through dependent code analysis techniques, such as data-flow analysis. We also plan to include semantic analysis which allows MiDas to capture changes in program semantics such as control-flow or data-flow to further improve the capability of MiDas. Finally, as the results are promising, MiDas can be utilized to automatically curate a benchmark of vulnerabilities that can be used to evaluate vulnerability detection systems. We plan to build a benchmark of vulnerabilities in future work.

Acknowledgement

This project is supported by the National Research Foundation, Singapore and National University of Singapore through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) office under the Trustworthy Comput- ing for Secure Smart Nation Grant (TCSSNG) award no. NSOE-TSS2020-02. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and National University of Singapore (including its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) office).

Xuan-Bach D. Le is supported by the Australian Government through the Australian Research Council's Discovery Early Career Researcher Award, project number DE220101057.

References

[1] "Log4shell: Rce 0-day exploit found in log4j 2, a popular java logging package," https://www.lunasec.io/docs/blog/log4j-zero-day.

[2] "Cve-2021-44228 – log4j 2 vulnerability analysis," https://www.randori.com/blog/cve-2021-44228.

[3] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem," *arXiv preprint arXiv:2201.03981*, 2022.

[4] S. E. Ponta, H. Plate, and A. Sabetta, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3175–3215, 2020.

[5] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 181–191.

[6] J. M. Gonzalez-Barahona, P. Sherwood, G. Robles, and D. Izquierdo, "Technical lag in software compilations: Measuring how outdated a software deployment is," in *IFIP International Conference on Open Source Systems*. Springer, Cham, 2017, pp. 182–192.

[7] A. Ihara, D. Fujibayashi, H. Suwa, R. G. Kula, and K. Matsumoto, "Understanding when to adopt a library: A case study on asf projects," in *IFIP International Conference on Open Source Systems*. Springer, Cham, 2017, pp. 128–138.

[8] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[9] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 269–280.

[10] B. Chinthanet, R. G. Kula, S. McIntosh, T. Ishio, A. Ihara, and K. Matsumoto, "Lags in the release, adoption, and propagation of npm vulnerability fixes," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–28, 2021.

[11] A. Zerouali, T. Mens, A. Decan, J. Gonzalez-Barahona, and G. Robles, "A multi-dimensional analysis of technical lag in debian-based docker images," *Empirical Software Engineering*, vol. 26, no. 2, pp. 1–45, 2021.

[12] "Owasp dependency-check," https://owasp.org/www-project-dependency-check/.

[13] H. J. Kang, T. G. Nguyen, B. Le, C. S. Păsăreanu, and D. Lo, "Test mimicry to assess the exploitability of library vulnerabilities," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 276–288.

[14] N. Imtiaz, A. Khanom, and L. Williams, "Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages," *IEEE Transactions on Software Engineering*, 2022.

[15] S. Pan, J. Zhou, F. R. Cogo, X. Xia, L. Bao, X. Hu, S. Li, and A. E. Hassan, "Automated unearthing of dangerous issue reports," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2022, pp. 834–846.

[16] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*, 2017, pp. 914–919.

[17] A. Sabetta and M. Bezzi, "A practical approach to the automatic classification of security-relevant commits," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 579–582.

[18] N. Truong-Giang, H. J. Kang, D. Lo, A. Sharma, A. Santosa, A. Sharma, and M. Yi Ang, "Hermes: Using commit-issue linking to detect vulnerability-fixing commits," in *The 2022 29th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2022.

[19] T. G. Nguyen, T. Le-Cong, H. J. Kang, X.-B. D. Le, and D. Lo, "Vulcurator: a vulnerability-fixing commit detector," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1726–1730.

[20] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *2021 36th IEEE/ACM Automated Software Engineering Conference (ASE)*, 2021.

[21] "Why do organizations trust snyk to win the open source security battle?" https://snyk.io/blog/why-snyk-wins-open-source-security-battle/.

[22] "Vulnerability database, securing your open source software depends on the industry's best data," https://www.mend.io/wp-content/media/2021/11/WhiteSource-Vulnerability-Database.pdf.

[23] J. Williams and A. Dabirsiaghi, "The unfortunate reality of insecure libraries," 2012.

[24] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries." in *NDSS*, 2019.

[25] J. Dai, Y. Zhang, Z. Jiang, Y. Zhou, J. Chen, X. Xing, X. Zhang, X. Tan, M. Yang, and Z. Yang, "BScout: Direct whole patch presence test for java executables," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1147–1164.

[26] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 383–387.

[27] J. Zhou, M. Pacheco, J. Chen, X. Hu, X. Xia, D. Lo, and A. E. Hassan, "Colefunda: Explainable silent vulnerability fix identification," 2023.

[28] Y. Chen, A. E. Santosa, A. M. Yi, A. Sharma, A. Sharma, and D. Lo, "A machine learning approach for vulnerability curation," in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, 2020, pp. 32–42.

[29] R. Ramsauer, L. Bulwahn, D. Lohmann, and W. Mauerer, "The sound of silence: Mining security vulnerabilities from secret integration channels in open-source projects," in *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2020, pp. 147–157.

[30] B. Wu, S. Liu, R. Feng, X. Xie, J. Siow, and S.-W. Lin, "Enhancing security patch identification by capturing structures in commits," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[31] "Iso/iec 29147:2018 information technology — security techniques — vulnerability disclosure," https://www.iso.org/standard/72311.html.

[32] A. D. Householder, G. Wassermann, A. Manion, and C. King, "The cert guide to coordinated vulnerability disclosure," Carnegie-Mellon Univ Pittsburgh Pa Pittsburgh United States, Tech. Rep., 2017.

[33] M. Pradel and S. Chandra, "Neural software analysis," *Communications of the ACM*, vol. 65, no. 1, pp. 86–96, 2021.

[34] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[35] K. Jesse, P. T. Devanbu, and T. Ahmed, "Learning type annotation: is big data enough?" in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1483–1486.

[36] M. Kazerounian, J. S. Foster, and B. Min, "Simtyper: sound type inference for ruby using type equality prediction," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[37] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 169–180.

[38] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 661–673.

[39] T.-D. Nguyen, T. Le-Cong, D.-M. Luong, V.-H. Duong, X.-B. D. Le, D. Lo, and Q.-T. Huynh, "Ffl: Fine-grained fault localization for student programs via syntactic and semantic reasoning," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 151–162.

[40] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[41] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 511–523.

[42] B. Lin, S. Wang, M. Wen, and X. Mao, "Context-aware code change embedding for better patch correctness assessment," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–29, 2022.

[43] X. Zhou, B. Xu, K. Kim, D. Han, T. Le-Cong, J. He, B. Le, and D. Lo, "Patchzero: Zero-shot automatic patch correctness assessment," *arXiv preprint arXiv:2303.00202*, 2023.

[44] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

[45] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 134–144.

[46] R. Polikar, "Ensemble learning," *Ensemble machine learning: Methods and applications*, pp. 1–34, 2012.

[47] G. Haixiang, L. Yijing, J. Shang, G. Mingyun, H. Yuanyue, and G. Bing, "Learning from class-imbalanced data: Review of methods and applications," *Expert systems with applications*, vol. 73, pp. 220–239, 2017.

[48] T. M. Khoshgoftaar, A. Fazelpour, D. J. Dittman, and A. Napolitano, "Ensemble vs. data sampling: Which option is best suited to improve classification performance of imbalanced bioinformatics data?" in *2015 ieee 27th international conference on tools with artificial intelligence (ictai)*. IEEE, 2015, pp. 705–712.

[49] X. Dong, Z. Yu, W. Cao, Y. Shi, and Q. Ma, "A survey on ensemble learning," *Frontiers of Computer Science*, vol. 14, pp. 241–258, 2020.

[50] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, 2020, pp. 1536–1547.

[51] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[52] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[53] "Codebert-base," https://huggingface.co/microsoft/codebert-base.

[54] C. Yu, G. Yang, X. Chen, K. Liu, and Y. Zhou, "Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 82–93.

[55] T. Le-Cong, H. J. Kang, T. G. Nguyen, S. A. Haryono, D. Lo, X.-B. D. Le, and Q. T. Huynh, "Autopruner: transformer-based call graph pruning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 520–532.

[56] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 505–509.

[57] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[58] T. Le-Cong, D.-M. Luong, X. B. D. Le, D. Lo, N.-H. Tran, B. Quang-Huy, and Q.-T. Huynh, "Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning," *arXiv preprint arXiv:2301.01113*, 2023.

[59] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[60] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[61] J. Han and C. Moraga, "The influence of the sigmoid function parameters on the speed of backpropagation learning," in *International workshop on artificial neural networks*. Springer, 1995, pp. 195–201.

[62] B. Xu, T. Hoang, A. Sharma, C. Yang, X. Xia, and D. Lo, "Post2vec: Learning distributed representations of stack overflow posts," *IEEE Transactions on Software Engineering*, 2021.

[63] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural cnn decoder for code generation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 7055–7062.

[64] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 34–45.

[65] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[66] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural networks*, vol. 18, no. 5-6, pp. 602–610, 2005.

[67] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1482–1493.

[68] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[69] Y. Fan, X. Xia, D. Lo, A. E. Hassan, Y. Wang, and S. Li, "A differential testing approach for evaluating abstract syntax tree mapping algorithms," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1174–1185.

[70] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," *arXiv preprint arXiv:1404.2188*, 2014.

[71] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach," *IEEE transactions on neural networks*, vol. 8, no. 1, pp. 98–113, 1997.

[72] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[73] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, "Deep just-in-time defect prediction: how far are we?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021, pp. 427–438.

[74] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (roc) curve." *Radiology*, vol. 143, no. 1, pp. 29–36, 1982.

[75] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[76] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 107–116.

[77] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886–894, 1996.

[78] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[79] X. Yu, K. E. Bennin, J. Liu, J. W. Keung, X. Yin, and Z. Xu, "An empirical study of learning to rank techniques for effort-aware defect prediction," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 298–309.

[80] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 157–168.

[81] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2823–2862, 2019.

[82] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.

[83] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "CC2Vec: Distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 518–529.

[84] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 2015, pp. 560–564.

[85] J. D. Brown *et al.*, *Understanding research in second language learning: A teacher's guide to statistics and research design*. Cambridge University Press, 1988.

[86] M. A. Pett, *Nonparametric statistics for health care research: Statistics for small samples and unusual distributions*. Sage Publications, 2015.

[87] D. Romano and M. Pinzger, "Using source code metrics to predict change-prone java interfaces," in *2011 27th IEEE international conference on software maintenance (ICSM)*. IEEE, 2011, pp. 303–312.

[88] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[89] L. Yu and H. Liu, "Feature selection for high-dimensional data: A fast correlation-based filter solution," in *Proceedings of the 20th international conference on machine learning (ICML-03)*, 2003, pp. 856–863.

[90] M. Massoudi, N. K. Jain, and P. Bansal, "Software defect prediction using dimensionality reduction and deep learning," in *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*. IEEE, 2021, pp. 884–893.

[91] S. K. Pandey, D. Rathee, and A. K. Tripathi, "Software defect prediction using k-pca and various kernel-based extreme learning machine: an empirical study," *IET Software*, vol. 14, no. 7, pp. 768–782, 2020.

[92] L. Goel, M. Sharma, S. K. Khatri, and D. Damodaran, "Defect prediction of cross projects using pca and ensemble learning approach," in *Micro-Electronics and Telecommunication Engineering*. Springer, 2020, pp. 307–315.

[93] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[94] M. Kondo, C.-P. Bezemer, Y. Kamei, A. E. Hassan, and O. Mizuno, "The impact of feature reduction techniques on defect prediction models," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1925–1963, 2019.

[95] T. Auld, A. W. Moore, and S. F. Gull, "Bayesian neural networks for internet traffic classification," *IEEE Transactions on neural networks*, vol. 18, no. 1, pp. 223–239, 2007.

[96] H.-L. Nguyen, Y.-K. Woon, W.-K. Ng, and L. Wan, "Heterogeneous ensemble for feature drifts in data streams," in *Pacific-Asia conference on knowledge discovery and data mining*. Springer, 2012, pp. 1–12.

[97] W. Li, M. Canini, A. W. Moore, and R. Bolla, "Efficient application identification and the temporal and spatial stability of classification schema," *Computer Networks*, vol. 53, no. 6, pp. 790–809, 2009.

[98] S. S. Kannan and N. Ramaraj, "A novel hybrid feature selection via symmetrical uncertainty ranking based local memetic search algorithm," *Knowledge-Based Systems*, vol. 23, no. 6, pp. 580–585, 2010.

[99] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, L. Cavallaro *et al.*, "Tesseract: Eliminating experimental bias in malware classification across space and time," in *Proceedings of the 28th USENIX Security Symposium*. USENIX Association, 2019, pp. 729–746.

[100] Y. Lyu, T. Le-Cong, H. J. Kang, R. Widyasari, Z. Zhao, X.-B. D. Le, M. Li, and D. Lo, "Chronos: Time-aware zero-shot identification of libraries from vulnerability reports," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023. [Online]. Available: https://arxiv.org/abs/2301.03944

[101] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 524–535.

[102] A. Kharkar, R. Z. Moghaddam, M. Jin, X. Liu, X. Shi, C. Clement, and N. Sundaresan, "Learning to reduce false positives in analytic bug detectors," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1307–1316.

[103] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O'Hearn, "Finding real bugs in big programs with incorrectness logic," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–27, 2022.

[104] F. Wilcoxon, *Individual comparisons by ranking methods*. Springer, 1992.

[105] M. Yan, X. Xia, D. Lo, A. E. Hassan, and S. Li, "Characterizing and identifying reverted commits," *Empirical Software Engineering*, vol. 24, pp. 2171–2208, 2019.

[106] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 82–101, 2020.

[107] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn, "Mining software defects: Should we consider affected releases?" in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 654–665.

[108] "Midas's replication package," https://github.com/soarsmu/midas.

[109] F. Chen, F. H. Fard, D. Lo, and T. Bryksin, "On the transferability of pre-trained language models for low-resource programming languages," in *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. IEEE, 2022, pp. 401–412.

[110] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "Spi: Automated identification of security patches via commits," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2021.

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

[112] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar, "Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 717–729.

[113] H. Zhong, X. Wang, and H. Mei, "Inferring bug signatures to detect real bugs," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 571–584, 2020.

[114] H. Zhong and X. Wang, "Boosting complete-code tool for partial program," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 671–681.