CoLeFunDa: Explainable Silent Vulnerability Fix Identification

Jiayuan Zhou*, Michael Pacheco*, Jinfu Chen*, Xing Hu†‡, Xin Xia‖, David Lo§ and Ahmed E. Hassan¶
* Centre for Software Excellence, Huawei, Toronto, Canada
† School of Software Technology, Zhejiang University, Ningbo, China
‖ Huawei, China
§ School of Information Systems, Singapore Management University, Singapore
¶ Software Analysis and Intelligence Lab (SAIL), Queen's University
{jiayuan.zhou1,michael.pacheco1,jinfu.chen1}@huawei.com,xinghu@zju.edu.cn, xin.xia@acm.org,
davidlo@smu.edu.sg, ahmed@cs.queensu.ca

***Abstract*—It is common practice for OSS users to leverage and monitor security advisories to discover newly disclosed OSS vulnerabilities and their corresponding patches for vulnerability remediation. It is common for vulnerability fixes to be publicly available one week earlier than their disclosure. This gap in time provides an opportunity for attackers to exploit the vulnerability. Hence, OSS users need to sense the fix as early as possible so that the vulnerability can be remediated before it is exploited. However, it is common for OSS to adopt a vulnerability disclosure policy which causes the majority of vulnerabilities to be fixed silently, meaning the commit with the fix does not indicate any vulnerability information. In this case even if a fix is identified, it is hard for OSS users to understand the vulnerability and evaluate its potential impact. To improve early sensing of vulnerabilities, the identification of silent fixes and their corresponding explanations (e.g., the corresponding common weakness enumeration (CWE) and exploitability rating) are equally important.**

However, it is challenging to identify silent fixes and provide explanations due to the limited and diverse data. To tackle this challenge, we propose *CoLeFunDa*: a framework consisting of a Contrastive Learner and FunDa, which is a novel approach for Function change Data augmentation. FunDa first increases the fix data (i.e., code changes) at the function level with unsupervised and supervised strategies. Then the contrastive learner leverages contrastive learning to effectively train a function change encoder, FCBERT, from diverse fix data. Finally, we leverage FCBERT to further fine-tune three downstream tasks, i.e., silent fix identification, CWE category classification, and exploitability rating classification, respectively. Our result shows that *CoLeFunDa* outperforms all the state-of-art baselines in all downstream tasks. We also conduct a survey to verify the effectiveness of *CoLeFunDa* in practical usage. The result shows that *CoLeFunDa* can categorize 62.5% (25 out of 40) CVEs with correct CWE categories within the top 2 recommendations.

I. Introduction

With the prevalent use of open-source software (OSS), OSS users must manage OSS vulnerabilities (e.g., sensing and fixing vulnerabilities) in time. Otherwise, OSS users will be exposed to large security risks which may lead to significant consequences. For example, with a late fix to the CVE-2017-5638 vulnerability [1], Equifax, one of the largest credit reporting agencies, suffered from a data breach, resulting in a loss of more than $650 million [2]. For better OSS vulnerability management, the Coordinated Vulnerability Disclosure (CVD) process [3] is widely adopted [4], [5], [6], [7], [8]. Following this process, a vulnerability should be fixed first and then disclosed publicly. This allows OSS users to start the remediation process immediately since the patch is already available once the vulnerability is disclosed. To limit the degree of dissemination of information related to the vulnerability, CVD also suggests that vulnerabilities should be fixed *silently*, for example, the commit message should not carry any information that may reveal the vulnerability.

Due to various reasons (e.g., the limited human resource or the long fix-to-integration release cycle [9]), the timing of public disclosure does not closely align with the fixed date. This time gap could vary from days to months (more than one week in median [9], [10]). For example, Log4Shell (CVE-2021-44228 [11]) is a vulnerability in Apache Log4J, a Java logging framework. An adversary can utilize Log4Shell to take complete control over the system by sending crafted requests, making the vulnerability easily exploitable. Given the popular use of Log4J in Java systems and the ease of its exploitability, Log4Shell is considered one of the most dangerous vulnerabilities. Log4Shell was first disclosed on Dec. 10, 2021 [11]. However, the corresponding fix [12] publicly existed 11 days earlier (Nov. 29, 2021). Such a time gap may provide a window of opportunity for exploitation, causing OSS users to be exposed to huge security risks during the time gap. Given the transparent nature of OSS, the malicious parties could easily uncover the fix and derive the corresponding vulnerability for developing and deploying exploits in advance. Hence, OSS users must sense silent fixes as early as possible to start the remediation process as soon as possible.

Moreover, we argue that identifying silent vulnerability fixes is just the first step and that an explanation of them is also important. The reason is that OSS users may not be experts on every OSS they use, making it challenging to understand and analyze silently fixed vulnerabilities. For example, "Restrict LDAP access via JNDI" is the commit message of the fix [12] for Log4Shell. Since no vulnerability information is provided

‡Corresponding author.

(e.g., no security keywords), it is hard for general OSS users to understand the fixed vulnerability. Due to such limited information, even if a tool could identify this fix, OSS users might ignore the fix because of the misunderstanding, making such an early warning ineffective. Hence, after receiving an alert of a silent fix, providing basic yet important information (e.g., the *CWE category* and the *exploitability rating*) of the corresponding fixed vulnerability could help OSS users to understand and evaluate the impact of the vulnerability.

Providing explanations for silent fixes is another challenge because of limited and diverse data. A previous study [10] showed that the median percentage of vulnerability fixes in OSS is only 0.35%. Moreover, fixed vulnerabilities are associated with a wide range of CWE categories [13], indicating the diverse causes, behaviors, and consequences of vulnerabilities. Given the extremely imbalanced class distribution combined with diverse patterns of the fix data, it is difficult for traditional machine learning and deep learning approaches to effectively learn information from the data. The current state-of-the-art, VulFixMiner [10], utilizes the added and removed code snippets from entire commits to identify silent fixes. As vulnerability fixing commits can contain mixed information from the whole commit, and along with the lack of code context information, it is hard for VulFixMiner to provide explanations for diverse fixes.

To tackle these challenges, we propose a framework, *CoLeFunDa* (Figure 1), which consists of a **co**ntrastive **le**arner and a novel **fun**ction change **da**ta augmentation component, *FunDa*. FunDa first increases the fix data (i.e., code changes) at the function-level. Then the *contrastive learner* [14], [15] effectively learns the representations of the diverse fix data by minimizing the distance between positives (i.e., similar data representations) and maximizing the distance between negatives (i.e., dissimilar data representations). More specifically, FunDa combines program slicing techniques [16], [17], [18], [19], [20] and CWE category information to augment function changes with unsupervised (i.e., the self-based) and supervised (i.e., the group-based) strategies (Phase 1). Next, the contrastive learner learns function-level code change representations from the diverse fix data and trains the function change encoder *FCBERT* (Phase 2). Based on the pre-trained encoder, CoLeFunDa is then fine-tuned creating three models, namely CoLeFunDa$_{fix}$, CoLeFunDa$_{cwe}$, and CoLeFunDa$_{exp}$, for three downstream tasks, i.e., automated silent fix identification, CWE category classification, and exploitability rating classification, respectively (Phase 3).

We conduct our experiments on 1,436 Java CVE patches from 310 OSS projects. For the silent fix identification tasks, CoLeFunDa$_{fix}$ is evaluated under a practical setting (i.e., the class distribution of the fixes is extremely imbalanced). The evaluation result shows that CoLeFunDa$_{fix}$ outperforms the state-of-the-art baseline VulFixMiner from 11% to 14% in terms of all effort-aware performance metrics (i.e., CostEffort@5%, CostEffort@20%, and $\mathbf{P_{opt}}$), indicating the effectiveness of CoLeFunDa$_{fix}$ in identifying more vulnerabilities with less manual inspection effort. In the CWE classification task, the evaluation result shows that CoLeFunDa$_{cwe}$ outperforms the best SOTA baseline by 6% to 72% in terms of macro AUC, macro precision, macro recall, and macro F1 score. For the exploitability rating classification task, the evaluation result shows that CoLeFunDa$_{exp}$ outperforms the best SOTA baseline by 24% to 54% in terms of macro AUC, macro precision, macro recall, and macro F1 score.

Since not every CVE is well maintained, some CVEs do not contain CWE category information (i.e. no-CWE CVEs). To help OSS users better understand and evaluate the potential risk of no-CWE CVEs, it is also practical to recommend the corresponding CWE categories. We apply CoLeFunDa$_{cwe}$ on the patches of 40 no-CWE CVEs and conduct a user study with 5 security experts. The result shows that the CWE of 37.5% (62.5%) of CVEs are correctly categorized within the top one (two) recommendations, indicating the effectiveness of CoLeFunDa$_{cwe}$ in practical usage.

In summary, this paper makes the following contributions:
(1) We advocate the importance of explainable vulnerability silent fix identification for better OSS vulnerability management. We propose a framework, CoLeFunDa, which significantly outperforms all state-of-the-art baselines in three explainable silent fix identification tasks.
(2) To the best of our knowledge, FunDa is the first approach for function-level code change data augmentation. Specifically, FunDa provides an unsupervised strategy for data augmentation that can be applied for large-scale general commit data directly.
(3) To the best of our knowledge, we propose the first unsupervised solution for function-level code change representation learning. The solution can be applied for training general function-level code change representation encoders, which is important for many software tasks (e.g., just-in-time defect prediction and commit message generation).
(4) We release the vulnerability fix dataset with the enhanced CVE information (e.g., the CWE categories and exploitability ratings) of our study [21].

II. Background

In this section, we briefly introduce contrastive learning, Common Vulnerabilities and Exposures (CVE), Common Weakness Enumeration (CWE), Common Vulnerability Scoring System (CVSS) and Exploitability Metrics.

A. Contrastive learning

Contrastive learning is widely used in Computer Vision [15] (CV) and Natural Language Processing domains [22] (NLP). The key characteristic of contrastive learning is data augmentation, which generates new data from existing data. By applying augmentation on a data point, two samples that are different but semantically similar are generated. Contrastive learning then tries to learn similar knowledge within the samples from the same data points, and learn the differences between samples generated from different data points. In the NLP domain, data augmentation commonly consists of manipulation of tokens (e.g., token reordering and similar

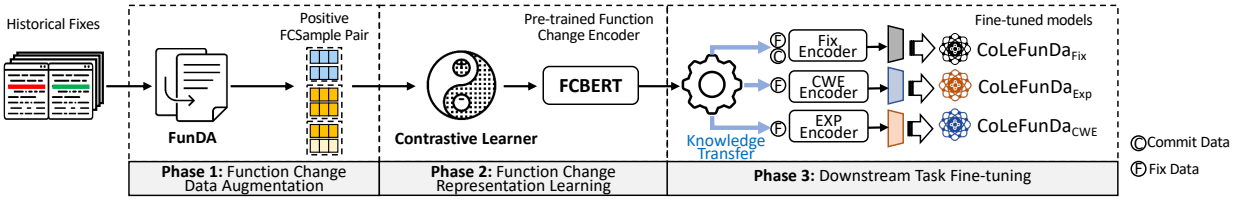


Fig. 1: Overall framework of CoLeFunDa.

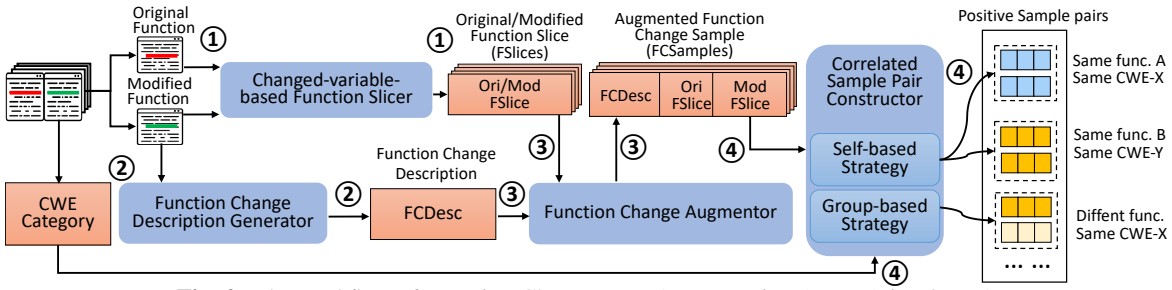


Fig. 2: The workflow of Function Change Data Augmentation (FunDa) in Phase 1.

token replacement). In the software engineering domain, prior studies focus on data augmentation of source code based on approaches from NLP. This includes sampling or augmentation strategies based on a compilation mechanism to generate source code samples [23] (e.g., code compression, identifier modification, regularization). These approaches achieve good performance in source code representation learning.

B. Common Vulnerabilities and Exposures (CVE) and common weakness enumeration (CWE)

Common Vulnerabilities and Exposure (CVE) database provides a reference-method for the disclosure, identification, and management of publicly known vulnerabilities. NVD [24] is a popular CVE database that provides enhanced vulnerability information such as CWE. CWE provides a dictionary of common weaknesses that can result in vulnerabilities in software or hardware. They include various details regarding several types of vulnerabilities. A CWE can be assigned to CVEs, providing a way to categorize and provides additional information about CVEs and their corresponding vulnerability. CVEs can be assigned multiple CWEs depending on the nature of the vulnerability, however CVEs without any assigned CWEs exist in NVD.

C. CVSS and Exploitability Metrics

CVSS helps define and categorize vulnerabilities based on their potential impact and risk. There are two CVSS versions, i.e., CVSS 2.0 and 3.0. CVSS version 3.0 was released in 2015, and is used for CVEs disclosed from then on. CVEs disclosed before 2015 have a CVSS 2.0 score [25] instead of the CVSS 3.0 version. Therefore, we use CVSS version 2.0 in our study. Exploitability is one of the base group metrics in CVSS, which is used to measure the risk of a vulnerability being exploited. The more easily a vulnerability can be exploited, the higher its exploitability score is. Therefore, the exploitability metric is valuable for practitioners to prioritize and fix the vulnerability.

III. Proposed Approach

The goal of CoLeFunDa is to learn function-level code change representations from diverse and limited vulnerability fixes to support explainable vulnerability silent fix early sensing. In this section, we first introduce the overall framework of CoLeFunDa (Figure 1). We then elaborate on the details of each phase. Finally, we explain the applications of CoLeFunDa.

CoLeFunDa has three phases: function change data augmentation, function change representation learning, and downstream task fine-tuning. In phase 1, we propose a novel approach, *FunDa*, to increase the amount of the vulnerability fix patch data. Specifically, for one function change from a patch, we augment it into a set of semantics-preserving function change samples (`FCSamples`). We then consider every two semantically-similar or functionality-similar FCSamples as a positive pair for contrastive learning in the next phase. In phase 2, we leverage contrastive learning to further pre-train a language model to recognize similar and dissimilar `FCSamples` and to generate the function change encoder, *FCBERT*. In phase 3, we leverage FCBERT to further fine-tune a silent fix identification model (*CoLeFunDa*$_{fix}$), a CWE classification model (*CoLeFunDa*$_{cwe}$), and an exploitability rating classification model (*CoLeFunDa*$_{exp}$), respectively.

A. Phase 1: Function Change Data Augmentation

Given the limited amount of vulnerability fixes in the dataset, it is a challenge to effectively learn the representations of function changes of the fixes, especially since the CWE categories of the corresponding fixes are diverse (see Section IV-B). To tackle this challenge, we propose FunDa for augmenting function change data. As shown in Figure 2, the workflow of our approach contains four steps. In step 1, for each function change, we generate function slices for original and modified functions (i.e., `OriFSlices` and `ModFSlices`), respectively. In step 2, we generate the function change description (`FCDesc`) for each function change. In step 3, we generate `FCSamples` by combining the `FCDesc`, `OriFSlices`, and

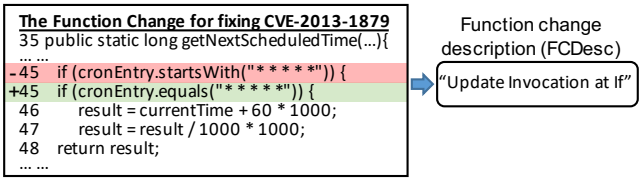


Fig. 3: An example of function change description (FCDesc) from the patch [29] for CVE-2013-1879 [30]. The patch fixed a Cross-site scripting vulnerability in Apache ActiveMQ.

`ModFSlices`. In step 4, we construct positive sample pairs by pairing every two semantically-similar or functionality-similar `FCSamples` with unsupervised (i.e., the *self-based*) and supervised (i.e., the *group-based*) strategies.

Step 1: Changed-variable-based function slicing. We leverage the program slicing technique to generate function slices (`FSlices`) for the original function and modified function, respectively. Since the changed code statements between the original function and modified function actually fix the vulnerability, we consider the changed variables in the changed code statement as our anchors for slicing. For slice generation, we focus on comprehensive slices, which merge aspects of both forward and backward slices [19], [20]. To compute such slices, we leverage both control flow graphs (CFGs) and data flow graphs (DFGs) [26], [17] since the combination of such graphs maintains the structural integrity of the original program, and extracts data relationships between variables in the program. We first leverage a source code parsing tool, namely TreeSitter [27], to generate CFG and DFG. Then for each anchor, we traverse and combine their respective control and data flow paths. Finally, for each anchor, we extract the corresponding code statements from these paths to form up a changed-variable based FSlices for the function. Figure 4 shows two examples of `OriFSlices` and `ModFSlices` for original and modified function, respectively.

Note that not every changed function contains changed variables. For example, some function changes are about function call renaming or operator changing. In this case, the function has no changed-variable based `FSlices` and we use the full function without slicing.

Step 2: Function change description (FCDesc) generation. Multi-modal pre-training can help text-based models learn the implicit alignment between inputs of different modalities, for example, between natural language and programming language. We introduce the `FCDesc` and consider it as complementary information to enhance the augmented function change samples in Step 3. We leverage the GumTree Spoon AST Diff tool [28] to generate a list of change operations for each original and modified function pair. The GumTree tool is capable of identifying insert and delete change operations, along with renaming or moving operations, providing detailed information of the change. Figure 3 shows an example of the `FCDesc` for the patch [29] that fixed a cross-site scripting vulnerability [30].

Step 3: Function change augmentation. In Step 1, for one function change, we derive `OriFSlices` and `ModFSlices` from its original and modified functions. In Step 2, we further

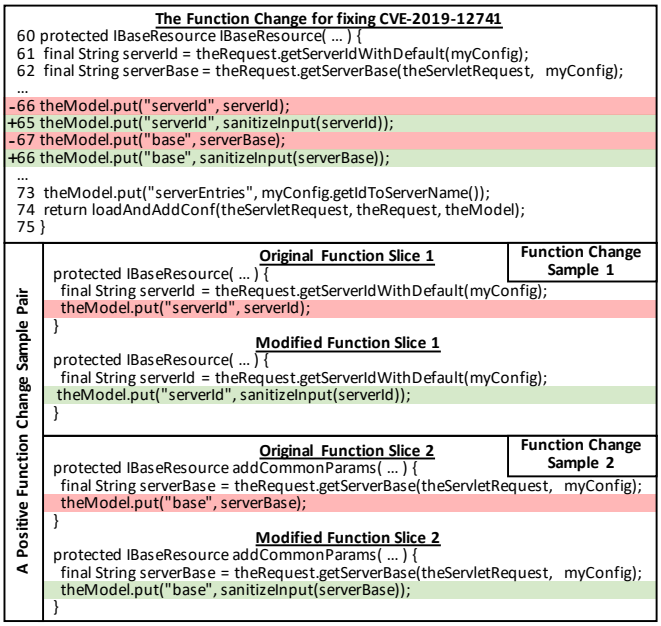


Fig. 4: An example of data augmentation on patch [31] for CVE-2019-12741 [32]. The patch fixed a potential security vulnerability by sanitizing input.

generate an `FCDesc` for the function change. In this step, we construct augmented `FCSamples` for the function change as:

$$FCSample_{ij} = FCDesc \oplus OriFSlice_i \oplus ModFSlice_j$$

where $\oplus$ is the concatenation operator, i and j are the i^{th} and the j^{th} `OriFSlices` and `ModFSlices`, respectively. To avoid the potential overfitting, for one function change, we randomly select at most four `FCSamples` for correlated sample pair construction (see step 4). Figure 4 shows two examples of `FCSamples` (without `FCDesc`).

Step 4: Positive Function Change Sample Pair Construction. With the `FCSamples` and the CWE category information of each function change (`FC_CWE`), the correlated sample pair constructor generates the positive `FCSamples` pairs. We consider two `FCSamples` as a positive function change sample pair if they are correlated (i.e., their semantic meanings are similar, or their functionality meanings are similar).

We construct positive pairs on two strategies: (1) The unsupervised **self-based** strategy is similar to the general data augmentation technique, which chooses two samples that are transformed from the same data instance. For example, two `FCSamples` belonging to the same function change are composed into one positive sample pair, as we consider they are semantically similar to each other. The function change which failed to generate `FCSamples` cannot be used in this strategy, as there are no samples that are directly semantically which exist. (2) The supervised **group-based** strategy, which leverages the `FC_CWE` information of function changes to construct positive pairs. For example, for a group of `FCSamples` belonging to different function changes which fix the same type of vulnerability (i.e., the same `FC_CWE`), we consider that the `FCSamples` within the same group are functionality similar to the other. Hence, such `FCSamples` can be used for creating a positive pair with any of the other

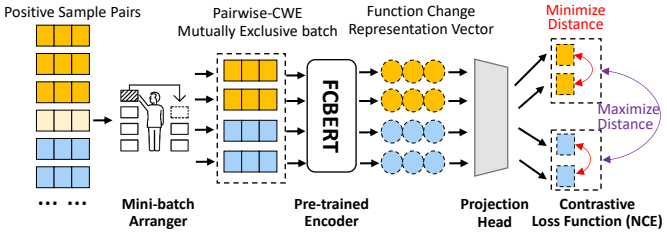


Fig. 5: The workflow function change representation learning (contrastive learner) in Phase 2.

samples in the group. One benefit of this strategy is that the function changes which failed to generate `FCSamples` (e.g., no changed variable for slicing) can still be paired with another functionality-similar sample.

The output of Phase 1 in Figure 1 illustrates three cases of constructed positive sample pairs. Note that we only apply the group-based strategy for commits that contain one function change only. This is because such a one-function commit change is confirmed to fix a specific type of vulnerability, as opposed to many function changes in a single commit.

B. Phase 2: Function change representation learning

To learn the representations of function changes, we employ a contrastive learner, which can learn data representation effectively by minimizing the distance between similar data (positives) and maximizing the distance between dissimilar data (negatives). Hence, with the constructed positive sample pairs from phase 1, the contrastive learning technology can effectively learn the function change representation from diverse vulnerability fixes. We first arrange inputs in a mini-batch where all positive pairs within the mini-batch are related to different CWE categories. In this way, any samples from one pair are negatively correlated to any samples from other pairs within a mini-batch. Next, we further pre-train an encoder (*FCBERT*), to encode a function change to its embedding representation vector. Then, a projection head maps the vector to the space where a contrastive loss is applied. As illustrated in Figure 5, contrastive learner contains the following four major components: mini-batch arranger, pre-trained encoder, projection head, and contrastive loss function. We introduce each of them as follows:

Mini-batch Arranger that arranges n correlated sample pairs from the candidate pairs where n is the $batchsize/2$. Since we want to minimize the distances of samples within the same correlated sample pair, and maximize the distances between samples from different sample pairs, the mini-batch arranger utilizes the `FC_CWE` to ensure each of the pairs in a single mini-batch corresponds to different CWE categories.

Pretrained Encoder is used to encode each of the `FCSamples` in the positive sample pairs to their corresponding function change representation vectors. We define our pretrained encoder as FCBERT, which uses the same architecture and weights from CodeBERT [33].

A Nonlinear Projection Head helps improve the representation quality of the layer before it [15]. We use an MLP with two hidden layers to project the function change representation vector to the space where a contrastive loss is applied.

Contrastive Loss Function is defined for maximizing the agreement of samples within the same correlated sample pair, and minimizing the agreement between samples from different sample pairs. We employ the Noise Contrastive Estimate (NCE) loss function [15] to compute the loss.

C. Phase 3: Downstream task fine-tuning

For better vulnerability early sensing, the identification of silent vulnerability fixes is only the first step. Additionally providing an explanation for identified silent fixes are important in the practical scenario. With this in mind, we design the following three downstream tasks: silent fix identification, CWE classification, and exploitability rating classification. As shown in Figure 1, FCBERT is used as a pre-trained model, in which the weights are transferred to initialize the *FixEncoder*, *CWEEncoder*, and *EXPEncoder* for three downstream task, respectively. We introduce each downstream task as follows:

Silent Fix Identification task: Similar to the prior work [10], the goal of this task is to predict the probability that a commit is for fixing a vulnerability. VulFixMiner [10] is the state-of-the-art baseline, which achieved the best performance in a real word setting (i.e., extremely imbalanced class distribution of the fixes). VulFixMiner chose CodeBERT [33] as the pre-trained model to fine-tune the task. In this fine-tuning task, we reproduce the VulFixMiner and replace CodeBERT with the FixEncoder. Except for the pre-trained model, we leave the architecture of VulFixMiner and input construction untouched. We refer CoLeFunDa$_{fix}$ as to the silent fix identifier. The input of the task is the general commit data and the patch data (i.e., the commits that fixed vulnerabilities). For every commit, CoLeFunDa$_{fix}$ outputs a score indicating the probability of the commit for fixing a vulnerability.

CWE Classification task: The goal of this task is to predict the probability that a given function change in a patch is for fixing a specific CWE category. The input of this fine-tuning task is the function-level patch data. More specifically, the function change description, the full original function, and the full modified function source code. The input is first encoded into a function change representation vector by *CWEEncoder*. The vector is then fed into a two-layer neural network to compute probability scores for each CWE category. Note that since one patch could be used for fixing a vulnerability assigned with multiple CWE categories, we consider this task as a multi-label classification task and employ the binary cross entropy as the loss function. We refer CoLeFunDa$_{cwe}$ as to the CWE classifier.

Exploitability Rating Classification task: The goal of this task is to predict the probability of the exploitability rating of the fixed vulnerability. The input and the process of fine-tuning in this task are identical to the CWE classification task, except for the loss function. Since one vulnerability only has one exploitability rating, we consider this task as a multi-class classification task and instead employ the cross entropy as the loss function. We refer CoLeFunDa$_{exp}$ as to the exploitability rating classifier.

D. Applications

Silent fix identification: Given a set of commits, CoLeFunDa$_{fix}$ computes the probability scores (i.e., the chance that the commit fixes a vulnerability) and outputs a list of commits ranked by their predicted probability.

CWE classification: Given a commit that is confirmed for fixing a vulnerability, for each function change within the commit, CoLeFunDa$_{cwe}$ computes a score for each CWE category as $CWE_jScore_i = CoLeFunDa_{cwe}(FC_i)$, where FC_i is the the i^{th} function change of the commit, and CWE_jScore_i is the score of the the j^{th} CWE category. The CWE scores of the commit is calculated as $CWE_jScore = \max_{1 \leqslant i \leqslant n} (CWE_jScore_i)$, where n is the #total function change within the commit. The CWE categories are ranked by their scores, indicating the probability of the commit fixing that specific category of CWE.

Exploitability rating classification: Given a commit that is confirmed for fixing a vulnerability, for each function change within the commit, CoLeFunDa$_{exp}$ computes the score for each possible exploitability rating as $EXP_jScore_i = CoLeFunDa_{exp}(FC_i)$, where EXP_jScore_i is the score of the j^{th} exploitability rating for the i^{th} function change. The commit-level scores of exploitability rating are calculated as $EXP_jScore = \max_{1 \leqslant i \leqslant n} (EXP_jScore_i)$, where n is the #total function change within the commit. The exploitability rating categories are ranked by their scores, indicating the probability of the commit fixing that specific exploitability rating.

Note that CoLeFunDa$_{fix}$, CoLeFunDa$_{cwe}$, and CoLeFunDa$_{exp}$ can be used either separately or sequentially. For better vulnerability early sensing, OSS users can integrate CoLeFunDa$_{fix}$, CoLeFunDa$_{cwe}$, and CoLeFunDa$_{exp}$ into an automatic pipeline to monitor OSS code repositories chronologically. With this pipeline, when a new code change is pushed to the public repository, CoLeFunDa$_{fix}$ can first identify whether the commit is for fixing a vulnerability. If it is, CoLeFunDa$_{cwe}$ and CoLeFunDa$_{exp}$ can further explain the relevant CWE category of the vulnerability together with the exploitability rating.

IV. Experiments

In this section, we introduce our research questions, dataset, baselines, evaluation metrics, and experiment setup. Finally, we show results for each research question.

A. Research Questions

In this paper, we aim to answer the following RQs:

RQ1: How effective is CoLeFunDa in early vulnerability sensing compared to the state-of-the-art baselines? We argue that for better vulnerability early sensing, the silent fix identification and the following vulnerability explanation are equally as important in real-world usage. With this as motivation, we evaluate the effectiveness of CoLeFunDa in this RQ in explainable vulnerability silent fix sensing across three tasks: silent fix identification, CWE classification, and the exploitability rating classification.

RQ2: How effective is each component of CoLeFunDa? FunDa is the foundation in CoLeFunDa. In this RQ, we conduct an ablation study to verify the effectiveness of the key design choices in FunDa: changed-variable-based function slicing, incorporating function change descriptions (*FCDesc*), and the group-based strategy for positive function code change sample pair construction. We experiment with three variants (i.e., CoLeFunDa w/o FCDesc, CoLeFunDa w/o Group, and CoLeFunDa w/o Slicing), each lacking one key design of CoLeFunDa.

B. Data Collection

Our dataset is leveraged primarily from a previous study [10] and consists of commit information from a set of Java OSS. We collect additional information to enhance this dataset to conduct our study. We describe the data collection process in more detail below.

Step 1. Collect Java CVEs, the corresponding fixes, and the not-fix commits from related repositories. We collect Java CVE data from the dataset used in a previous study [10] available on Zenodo [34]. The Java portion of this dataset includes 839 CVEs, which correspond to 1,436 vulnerability fixes, and spans 310 OSS projects. The dataset also includes 839,682 not-fix commits, resulting in a total of 841,118 Java commits and the distribution of the vulnerability fixes is extremely imbalanced (1.71%).

Step 2. Retrieve function level changes from vulnerability fixes. We use the commit IDs from vulnerability fixes in Step 1 to collect function-level changes performed in these fixes. We leverage PyDriller [35] to collect the original and modified versions of the code for the changed functions, which we define as *function change pairs*. In total, we collect 8,423 function code pairs from the included patches.

Step 3. Enhance CVE with CWE category and exploitability information. We collect the CWE and exploitability information for each CVE from NVD [36]. We assign CWE and exploitability information for patches according to the information from their corresponding CVEs. In total, we collect 89 types of CWEs. There are approximately 92.3% (554 out of 600) CVEs that have only one CWE category assigned. We also observe that 52.8% of the CWE categories have less than 3 CVEs, indicating how scattered and insufficient the knowledge of different CWE categories is, and how it can be difficult to learn.

Step 4. Collect CWE hierarchy information. Finally, we collect the hierarchy information for CWE categories made available by Mitre through the Research Concepts [13] view. This hierarchical organization provides a method for categorizing CWEs based on their characteristics and defining their inter-dependencies. We compute a tree representation of all CWE categories in our dataset based on this hierarchy, where each node represents a CWE, and each edge represents a parent-child relationship between two CWEs. The *ancestor CWE category* is the top-level parent of a CWE in the hierarchy tree.

C. Data Preprocessing

In this subsection, we describe the steps taken to preprocess our data. For the CWE classification and exploitability rating classification task, the dataset only contains vulnerability fixes. For the silent fix identification task, the dataset includes the non-vulnerability fixing commits.

Step 1. Relabel CVEs with their ancestor CWE categories. In our dataset, 52.8% of CWEs only correspond to one or two CVEs, making it difficult to train models with such limited data for these CWEs. Given the hierarchical relationship of CWE categories, the CWEs of CVEs belonging to the same parent CWE category are similar at a high level. Hence, we consider patches from the CVEs that have the same ancestor CWE category to contain similar knowledge. We use the CWE hierarchical relationship information from the CWE-1000 Research Concepts to locate and relabel the *ancestor CWE category* for each CVE. This results in a total of 22 ancestor CWE categories.

Step 2. Clean the dataset. We follow a series of steps to filter our dataset. Firstly we remove CVEs that have infrequent (i.e., less than 5) ancestor CWE categories. After cleaning, we have 11 ancestor CWE categories in total, and 40 CVE without CWE information. We keep commits corresponding to the unlabeled CVEs in a separate dataset.

Step 3. Categorize exploitability rating information. We adopt the CVSS 3.0 severity rating criteria [37] to automatically categorize the exploitability score into four rating categories: low (0.1-3.9), medium (4.0-6.9), high (7.0-8.9), and critical (9.0-10.0). After categorizing, there are 22, 46, 376, and 360 CVEs rated with low, medium, high, and critical exploitability, respectively. Since the CVEs with low and medium exploitability are less dangerous, we merge these two categories into one category, namely the "low or medium" category. Therefore, there are 68 CVEs with low or medium exploitability ratings.

Step 4. Ancestor-CWE-based stratified data splitting. We split our dataset into training (60%), validation (20%), and test (20%) datasets. To balance the distribution of the ancestor CWE categories across the three datasets, we conduct stratified splitting based on the ancestor CWE category of patch data. Table I describes #functions and #commits belonging to each ancestor CWE in each dataset. To better utilize the patch data, we keep the CVEs only with rare ancestor CWE categories (which are filtered out in Step 2) in the training dataset.

Note that we use training data for function change representation learning and all three datasets for the CWE classification and the exploitability rating tasks. For the silent fix identification task, we also include the not-fix commits from prior work [10] (see step 1 in Section IV-B). To make a fair comparison with the state-of-the-art, VulFixMiner, we reuse the same test dataset in [10] for evaluation. We further clean the test data by removing the commits which also exist in our training or validation dataset and re-evaluate VulFixMiner on this cleaned test dataset.

Step 5. Data augmentation. Finally, we apply the approach described in Section III-A to augment function changes in the training dataset. Note that we only use training data for data augmentation. Hence, CoLeFunDa does not take advantage of test data during the evaluation process. As a result, we generated 2,636 augmented function change samples from the training dataset with 1,235 function changes. We further construct 2,025 and 15,468 positive pairs from group-based and self-based strategies, respectively.

D. Baselines

For the silent fix detection task, we only choose VulFixMiner as our baseline as it outperforms several state-of-the-art baselines. We further adapt VulFixMiner to the CWE classification and exploitability rating classification tasks, and also select three widely-used deep learning models, TextCNN, BiLSTM, and Transformer, as baselines.

VulFixMiner [10] is a transformer-based model. It learns commit-level code change representation to identify silent vulnerability fixes. This model achieves SOTA performance compared to several baselines in a real-world scenario involving an imbalanced test set. We also adapt VulFixMiner to the CWE and exploitability rating classification tasks (i.e., VulFixMiner$_{cwe}$ and VulFixMiner$_{exp}$, respectively) by leveraging its commit-level code change encoder.

TextCNN [38] is a Convolutional Neural Network (CNN), introducing the application of the CNN for natural language-related tasks. TextCNN is used, and in some cases achieves SOTA performance, on several NLP and software engineering tasks [39], [40], [41].

BiLSTM [42], [43] consists of two Long short-term memory (LSTM) networks [44], which are RNN-based models used to represent the sequential information of code. In BiLSTM, one LSTM consumes input in a forward manner, and the other in a backward manner. BiLSTM is applied throughout various software engineering studies [45].

Transformer [46] is a deep learning language model consisting of encoder-decoder(s) and multi-head attention layers. This model established the use of pre-trained models, and its architecture continues to be used for a variety of programming language processing tasks [47], [48].

E. Evaluation Metrics

We use a combination of effort-based metrics and canonical classification metrics to evaluate downstream tasks. Similar to the prior work [10], we evaluate the silent fix identification task with AUC and effort-ware metrics (i.e., CostEffort@L and P_{opt}). We evaluate CWE classification and Exploitability rating classification tasks with Precision, Recall, F1, and AUC.

AUC, the area under the receiver operating characteristic (ROC) curve [49] measures the prediction performance of the model for all possible classification thresholds (i.e., from 0 to 1). Compared to threshold-dependent metrics like precision, recall, and F1 score, the AUC is robust in quantifying the discriminative capability of a classifier, especially in imbalanced class distributions. This is due to its insensitivity toward imbalances in class distirbutions [50], [51], [52]. Generally, an

TABLE I: Data description of #Functions, #Commits after ancestor-CWE-based stratified data splitting at commit-level.

	CWE-16	CWE-19	CWE-254	CWE-264	CWE-284	CWE-310	CWE-399	CWE-664	CWE-691	CWE-693	CWE-707
Train	11, 3	28, 10	17, 7	95, 27	137, 44	45, 10	9, 5	574, 163	19, 10	75, 17	229, 85
Val	6, 1	13, 3	6, 2	33, 10	19, 11	16, 3	1, 1	194, 58	12, 5	29, 7	81, 27
Test	13, 1	22, 3	3, 2	30, 9	49, 14	13, 4	22, 2	175, 56	4, 3	28, 7	77, 29
Total	30, 5	63, 16	26, 11	158, 46	205, 69	74, 17	32, 8	943, 277	35, 18	132, 31	387, 141

AUC between 0.7 and 1 defines a classifier to have achieved acceptable performance [53].

CostEffort@L is an effort-aware metric used to evaluate VulFixMiner [10]. VulFixMiner is evaluated on how many vulnerabilities fixing commits it can identify within a limited inspection effort, defined as the modified lines of code (LOC). As a result, *CostEffort@L* is defined as the proportion of inspected vulnerability fixes among all the actual vulnerability fixes when L (LOC) of all commits are inspected. To evaluate our models using the same downstream task, we also compute *CostEffort@5%* and *CostEffort@20%* in our study.

$P_{opt}$the normalized version of the effort-aware performance metrics [54], is also used to evaluate VulFixMiner. This metric is a widely used effort-aware performance metric in defect prediction [55], [56], [57], [58]. We also calculate this metric when 5% and 20% of the LOCs are inspected, denoted as P_{opt}@5 and P_{opt}@20, respectively [10].

Precision, Recall, F1 are canonical classification metrics in defect prediction studies [59], [60], [61], [62], [63]. Our goal is to evaluate our models in classification tasks, which involve predicting between multiple classes. In such tasks, precision, recall, and F1 evaluate models in terms of how well they predict instances of each of the classes. We calculate the *macro* version of these metrics, which are suitable for multi-class and multi-label tasks.

F. Experiment Setup

For CoLeFunDa, an input consists of natural language (i.e., the function change description) and the code change (i.e., the original function and the modified function). The maximum input length of natural language and code change are 128 tokens and 256 tokens, respectively. We apply padding or truncation on inputs to keep the same length (i.e., 384 tokens). In the contrastive learner (Section III-B), we set the size of the nonlinear projection head to 768. We use Adam [64] with shuffled mini-batches. The learning rate of Adam is 1e-5 and the batch size is 8. The temperature parameter of the NCE loss is set to 0.7. We train FCBERT for 15 epochs and select the model with the lowest NCE loss. In the downstream tasks (Section III-C), for the silent fix identification, we reproduce VulFixMiner with the same hyperparameter setting. For the CWE and exploitability rating classification tasks, the sizes of the hidden layers in the two-layer neural network are both set to 768.

As for the baseline in the silent fix identification task, we contact the author of VulFixMiner for the model and evaluate the performance on the cleaned test data (see Step 4 in Section IV-C). In the CWE and Exploitability rating classification tasks, the input is a combination of the original function and the modified function, and the maximum length of the input is 256 for all baselines. For VulFixMiner$_{cwe}$ and VulFixMiner$_{exp}$, we follow [10], constructing two one-layer neural network classifiers for CWE and exploitability classification, respectively. For TextCNN baseline, we follow the majority of hyperparameters of prior work [39], and set the number of epochs to 15, embedding dimension length to 256, the kernel size to 3, and the number of filters to 100. For the transformer baseline, we set the same size of hidden states as CoLeFunDa. As for the other hyperparameters, we follow the same settings in [46]. For the BiLSTM baseline, we follow [10], constructing a network with an unrolling length of 32 and a hidden unit size of 256. Our experiments use 8 cores of Intel Xeon 2.7GHz CPU and a V100-32GB GPU [65].

TABLE II: Performance of CoLeFunDa$_{fix}$ and VulFixMiner in the silent fix identification task.

Model name	CostEffort (5%,20%)	P_{opt}(5%,20%)	AUC
VulFixMiner	0.52, 0.65	0.45, 0.56	0.79
CoLeFunDa$_{fix}$	**0.59, 0.72**	**0.51, 0.63**	**0.80**

G. Research Questions and Results

RQ1: CoLeFunDa vs. VulFixMiner

To determine how effective CoLeFunDa is in explainable vulnerability early sensing compared to other SOTA models, we use three downstream tasks: silent fix identification, CWE classification, and exploitability rating classification. We explain the tasks and their results in more detail below.

Silent Fix Identification. One strenuous way to identify silent fix is to inspect every commit manually. In this task, the goal is to identify silent fixes with less effort earlier. Table II presents the evaluation results of CoLeFunDa$_{fix}$ and its baseline, i.e., VulFixMiner. CoLeFunDa$_{fix}$ outperforms VulFixMiner in all metrics, especially on the effort-aware metrics, CostEffort and P_{opt}. CoLeFunDa$_{fix}$ achieves an improvement of 14%, 11%, 13%, and 11% in CostEffort@5%/10%, and P_{opt}@5%/20%, respectively. The slightly improved AUC and significantly improved effort-aware metrics indicate that CoLeFunDa$_{fix}$ maintains the same high discriminative capability in such an extremely imbalanced dataset, while dramatically decreasing the inspection effort of identifying the vulnerability fixes for developers.

CoLeFunDa$_{fix}$ shows the highest performance in detecting the fixes associated with CWE-264 (Permissions Privileges and Access Controls) and CWE-310 (Cryptographic Issues), with 0.71 and 0.67 recall, respectively. However, the approach almost failed in detecting fixes associated with CWE-69 (Protection Mechanism Failure) and CWE-691 (Insufficient Control Flow Management). We observe that the fix complexity (i.e., #File_involved) for CWE-693 and CWE-691 is 2 on average, which is higher than that for CWE-264 and CWE-310 (1.35 on average). Since CoLeFunDa$_{fix}$ is trained at function-

TABLE III: Performance of CoLeFunDa$_{cwe}$ and SOTA in the CWE classification task.

Model name	Precision$_{macro}$	Recall$_{macro}$	F1$_{macro}$	AUC
BiLSTM	0.09	1.0	0.15	0.73
TextCNN	0.07	0.41	0.09	0.63
VulFixMiner$_{cwe}$	0.08	0.41	0.09	0.38
Transformer	0.27	0.39	0.29	0.80
CoLeFunDa$_{cwe}$	**0.52**	**0.54**	**0.50**	**0.85**

TABLE IV: Performance of CoLeFunDa$_{exp}$ and SOTA in vulnerability exploitability rating classification task.

Model name	Precision$_{macro}$	Recall$_{macro}$	F1$_{macro}$	AUC
BiLSTM	0.32	0.43	0.37	0.68
TextCNN	0.39	0.47	0.41	0.76
VulFixMiner$_{exp}$	0.38	0.13	0.16	0.65
Transformer	0.41	0.54	0.44	0.67
CoLeFunDa$_{exp}$	**0.63**	**0.67**	**0.64**	**0.85**

level and may not learn the semantic relationship among fixes across files, hence, performance is poor in detecting complex fixes. In the future study, we plan to enhance CoLeFunDa$_{fix}$ by considering such cross-file fixes (e.g., utilize static analysis techniques to construct function dependency and merge co-change functions as one function).

CWE Classification. Table III summarizes the performance results of CoLeFunDa$_{cwe}$ and the SOTA baselines for this task. We observe that CoLeFunDa$_{cwe}$ significantly outperforms the BiLSTM model in all metrics except for the macro recall. On manual inspection, we find that the BiLSTM predicts all classes for each instance. This causes a macro recall of 1, and a very low macro precision. Compared to TextCNN and VulFixMiner$_{cwe}$, CoLeFunDa$_{cwe}$ outperforms both by substantial margins in all metrics.

In general, Transformer model performs the best among all baselines. Compared to the Transformer, CoLeFunDa$_{cwe}$ has an improvement of 93%, 39%, 72%, and 6% in terms of macro precision, macro recall, macro F1, and AUC, respectively. One possible explanation is that contrastive learner helps FCBERT effectively learn knowledge from the diverse fix data. So that FCBERT can encode function change into better representation vectors than Transformer. We further discuss FCBERT and Transformer in Section V-B.

Note that the CWE classification task is a multi-label classification task and the dataset is imbalanced, thus the F1-score of CoLeFunDa$_{cwe}$ is good enough to predict types for a vulnerability. In the test dataset, there are four minor CWE categories that only have no more than four fixes (i.e., CWE-16, CWE-19, CWE-254, and CWE-310). CoLeFunDa$_{cwe}$ achieves a minimum F1 score of 0.67 for each of them, indicating the effectiveness of CoLeFunDa$_{cwe}$ in classifying CWE categories even when the data sizes are small.

We also further evaluate the effectiveness of CoLeFunDa$_{cwe}$ in the real world scenario in Section V-A.

Vulnerability Exploitability Rating Classification. In this task, the performance results of CoLeFunDa$_{exp}$ and the SOTA baselines are summarized in Table IV. Among baselines, Transformer model performs the best. However, compared to the Transformer model, CoLeFunDa$_{cwe}$ achieves an improvement of 54%, 24%, 46%, and 27% in macro precision, macro recall, macro F1, and AUC, respectively. VulFixMiner$_{exp}$ performs the worst, which can be due to: (1) it only utilizes the code changes and ignores the code context information; (2) it mixes the information from the entire commit, which we see in some not-fix code changes are noisy. CoLeFunDa$_{exp}$ outperforms all baselines in all metrics by a substantial margin. The results shown in Table IV illustrates that CoLeFunDa$_{exp}$ is effective for the vulnerability exploitability rating classification task.

In terms of the classification performance in each exploitability class, CoLeFunDa$_{exp}$ outperforms all the baselines in terms of classification performance in each exploitability class. For example, the Transformer model, which performs the best among baselines, achieves 0.21, 0.49, and 0.64 in classifying the "low or medium", "high", and "critical" categories respectively. As a comparison, CoLeFunDa$_{exp}$ achieves an improvement of 48% (0.31), 41% (0.69), and 8% (0.69) respectively, indicating the ability of CoLeFunDa$_{exp}$ in classifying exploitability class.

In summary, compared to the baselines CoLeFunDa can support explainable silent fix sensing with the lowest manual inspection effort in identifying silent fixes. It also provides the best performance in categorizing the CWE category, and exploitability rating for the fixed vulnerabilities.

RQ2: Impact of FunDa

Table V shows the performance of CoLeFunDa and its three variants. We construct variants by removing function change description (*w/o FCDesc*), group-based strategy for positive sample construction (*w/o Group*), and generating function slices for self-based strategy (*w/o Slice*), respectively. For the CWE classification task, we can find that CoLeFunDa has similar performance in the *w/o Group* setting. However, the macro precision, macro recall, and macro F1 decrease when removing the FCDesc and the Slicing components. Thus, the FCDesc and the Slicing contribute to the performance of CoLeFunDa on the CWE classification. Compared to the Slicing, the FCDesc and the group-based strategy contribute more to the vulnerability exploitability rating classification task and silent fix detection task. Hence, the design of each component effectively improves the overall performance.

V. Discussion

A. Classifying CWE Categories for No-CWE CVEs

In practice, not all CVEs are assigned CWE category information. CVEs without CWEs lack crucial information that practioners can use to quickly understand the CVE. CoLeFunDa$_{cwe}$ can be used to directly classify the CWE category for the no-CWE CVEs. Therefore, we conduct a user study to further evaluate the correctness of the CWE classification provided by CoLeFunDa.

Experimental Tasks. We created tasks using all 40 no-CWE CVEs from Step 2 in Section IV-C, 16 of which are considered to be of critical severity. For each CVE (task), CoLeFunDa$_{cwe}$ outputs the probability of each ancestor CWE category corresponding to the given CVE. We ranked the

TABLE V: Performance of CoLeFunDa and its variants.

Model name	CWE Classification				Exploitability Rating Classification				Silent Fix Detection		
	Precision_{macro}	Recall_{macro}	F1_{macro}	AUC	Precision_{macro}	Recall_{macro}	F1_{macro}	AUC	CostEffort (5%,20%)	P_{opt} (5%,20%)	AUC
w/o FCDesc	0.48	0.50	0.46	0.84	0.45	0.60	0.51	0.80	0.57, 0.66	0.49, 0.59	0.78
w/o Group	0.51	0.50	0.49	0.83	0.45	0.64	0.53	0.79	0.50, 0.62	0.46, 0.55	0.75
w/o Slicing	0.46	0.49	0.44	0.85	0.48	0.64	0.55	0.79	0.57, 0.68	0.50, 0.60	0.78
CoLeFunDa	**0.52**	**0.54**	**0.50**	**0.85**	**0.63**	**0.67**	**0.64**	**0.85**	**0.59, 0.72**	**0.51, 0.63**	**0.80**

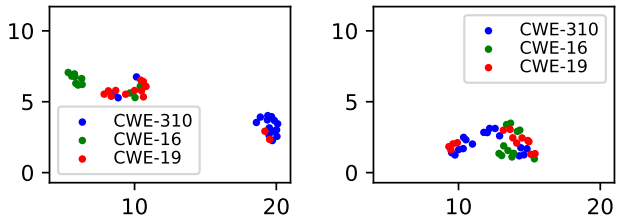
category of ancestor CWE based on their probabilities. Each participant was given eight CVEs with the classification and ranking results provided by CoLeFunDa_{cwe}. We randomized the order of the CVEs for each participant. For each task, we presented each participant with the CVE URL, the commits that fix the given CVE, and the top five recommended ancestor CWE categories provided by CoLeFunDa_{cwe}. The participants were needed to read the CVE description and the commit code changes and attempt to understand the true ancestor CWE category of the given CVE. The participants were then asked to answer (1) what is the exact ancestor CWE category of the CVE and (2) whether the true category is in the recommendations provided by our approach.

Participants. We invited five security experts from a prominent IT enterprise with at least five years of experience in software security to participate in our user study.

Results. Overall, our approach provides an effective approach for CWE category classification. First, practitioners confirm that 92.5% of the CVEs involved with our user study are classified into the correct CWE category recommended by CoLeFunDa_{cwe}. Second, 37.5% (62.5%) of CVEs are categorized into the top one (two) recommendations. This implies that the top two recommended CWEs provided by CoLeFunDa_{cwe} often cover the true CWE category. On the other hand, we also find that our model CoLeFunDa_{cwe} misclassified three CVEs. Particularly, participants fail to label the CWE category of CVE-2012-3439 due to the large code change of the commit. For the remaining two CVEs, CVE-2015-0263 is labeled as CWE-707 due to the code change fixing URL content. The reason given by the participant highly corresponds to CWE-264 provided by CoLeFunDa_{cwe}, a weakness related to permissions and privileges used to perform access control. The last misclassified CVE, CVE-2017-3156, is labeled with CWE-697 (Incorrect Comparison) while classified into CWE-691 (Insufficient Control Flow Management) by CoLeFunDa_{cwe}. The commit used to fix CVE-2017-3156 changes source from "equals" to "isEqual". Such a change can be a condition change resulting in different controls.

B. Visualization for Patch Function Change Representation

To understand and explain why the vectors produced by FCBERT are better than others, we visualize the learned representation space of vulnerability fixes. We randomly select three CWE categories (i.e., CWE-16, CWE-19, and CWE-310) and then use FCBERT to encode the function changes of the selected CWE categories from the validation dataset. We then apply UMAP [66] to reduce the dimensionality of the vectors into two-dimensional space. Representations of function changes of the same type of CWE are illustrated with



(a) FCBERT **(b)** Transformer

Fig. 6: Visualizations of function change representation learned by *FCBERT* and Transformer.

the same color. Note in Figure 6a, FCBERT groups function changes of the same type of CWE closely together. Conversely, the representations learned by the transformer show more overlap between different types of CWE (see Figure 6b). This implies that FCBERT can learn the representation of patch function changes better than the transformer in terms of CWE classification task.

C. Usage scenario

One benefit of CVD is to provide maintainers a chance and time to fix the vulnerability before the impact grows. Due to various reasons (e.g., long fix-to-integration release cycle), when the fixes are made, the vulnerabilities are not disclosed in time (one-week latency in median). CoLeFunDa could assist OSS users to sense the fixes at an early stage, so they could have more time in defending against potential attacks. Cybersecurity is always an arms race and often a two-edged sword. Different from solutions used by malicious parties, our solution will be open and is created with the intent to protect. Our solution is the same as many other solutions (e.g., Software composition analysis tools) that identify vulnerabilities. Those can be used by malicious parties and OSS users. However, such research is still valuable as it empowers OSS users to defend against potential attacks.

VI. Threats To Validity

Internal validity. Ont threat is that we conduct our study by leveraging function-level changes from vulnerability fixes (commits) and it is possible that not all function changes within a commit are for fixing the vulnerability and such function changes will introduce bias. To reduce the bias, we filter out the commits which have more than four function changes (Section IV-C). Additionally, in our supervised augmentation strategy, we only consider two function changes as a functionality similar pair when they are for fixing the same CWE and they are the only function change in the corresponding commits. Future studies should investigate different thresholds and filter approaches to reduce the bias caused by noisy commits.

One threat is the potential bias caused by the manual CWE classification in the user survey. To conduct a more reliable user study, the participants we invited are all senior security experts with more than 5 years of industry experience. To reduce the likelihood of biased answers, we provide the "other" option, so that the participant can provide an out-of-the-list CWE class according to their expertise. From the responses we received, three participants provided their own answers by choosing the "other" option.

Another threat is the quality of the FCDesc (i.e., the function change description). Since GumTree generates descriptions based on the edit script of the diff, the FCDesc generated by GumTree can only contain the node types and the change types, which are general. For example, it is possible that several fixes which fixed different CWEs, respectively, have the same the FCDesc generated by the GumTree. In such cases, the model might be confused and cannot better learn the corresponding knowledge of the fixes. We observe that there are 14% of vulnerability fixes have the same FCDesc with at least one other fix associated with different CWEs. One possible way to improve the quality of FCDesc is to utilize the human-annotated function change information. We encourage future studies to extract FCDesc for a specific function change from the commit message.

External validity. We conduct our experiment on Java patches and our approach may not generalizable to other programming languages. To mitigate the threat, to implement the key designs (e.g., function change description generation and program slicing) in our approach, we choose *Gumtree AST Spoon Diff* and *TreeSitter*. Both tools support a wide range of programming languages (i.e., 42 and 13, respectively) [67].

VII. Related Work

Contrastive learning in software engineering. Much prior research has proposed the application contrastive learning to better learn representations of code [23], [68], [69], [70]. Jain et al. [23] proposed *ContraCode* that leveraged code compression, identifier modification, and regularization to generate functionally equivalent programs. Ding et al. [68] presented *BOOST*, which uses contrastive learning to learn code representations based on code structure. Wang et al. [69] proposed *SYNCOBERT* that used symbolic and syntactic properties of the source code to better learn code representations. Bui et al. [71] proposed *Corder*, which used different program transformation operators (e.g., variable renaming and permutation of statements) to generate different program variants while preserving semantics. These studies have shown that contrastive learning can be used to learn function-level source code representations effectively on unlabelled datasets. In contrast to these studies, our goal is to learn the function-level code change representation. We propose a novel data augmentation approach for function changes. Our approach combines the programming language slicing techniques and CWE category information to generate semantic-meaning similar or functionality-meaning similar function-level code changes.

Early vulnerability sensing. Early vulnerability sensing can significantly reduce resources required to locate and fix the vulnerability and avoid vulnerability exploitation. Zhou et al. [10] proposed VulFixMiner, which is a Transformer-based approach, capable of extracting semantic meaning from commit-level code changes to identify silent fixes. VulFixMiner incapable of providing an explanation for the identified fixes since it only utilizes the code change information which is insufficient. DeepCVA [72] has been proposed to assess commit-level software vulnerabilities to understand the impact and severity of these vulnerabilities. DeepCVA used an attention-based convolutional neural network to extract features of code and surrounding context from vulnerable commits and assess seven metrics of software vulnerability. DeepCVA is incapable of early vulnerability sensing. Different from these studies, we provide CoLeFunDa for explainable silent vulnerability early sensing. With the identified silent fixes, we can further provide OSS users with two basic yet important explanations, the CWE category and the exploitability rating of the fixed vulnerability.

VIII. Conclusion and Future Work

In this work, we propose a framework, *CoLeFunDa*, for explainable vulnerability silent fix identification. CoLeFunDa leverages a novel function change augmentation, FunDa, and contrastive learning to learn function-level patch representations from limited and diverse patch data. CoLeFunDa is then fine-tuned for downstream tasks (i.e., silent fix identification, CWE category classification, and exploitability rating classification), to provide explainable automated silent fix identification. We construct CoLeFunDa on 1,436 CVE patches and 839,682 non-vulnerability fixing commits belonging to 310 Java OSS projects. Evaluation results show that CoLeFunDa outperforms the SOTA baseline models in all the downstream tasks. We further applied CoLeFunDa$_{cwe}$ to recommend a CWE category for 40 real-world no-CWE CVEs, and conduct a user study with 5 security experts to verify the correctness of CoLeFunDa$_{cwe}$ outputs. The results show that 62.5% of CVEs are classified correctly within the top 2 recommendations, indicating the effectiveness of CoLeFunDa$_{cwe}$ in the practical setting.

In future work, to provide a more comprehensive explanation for identified silent fixes, we plan to extend our downstream tasks to predict CVSS vulnerability metrics (e.g., the impact score and the severity level). We also plan to generalize our approach to more programming languages like Python and C/C++.

Acknowledgment

This project is supported by the National Key Research and Development Program of China (No. 2021YFB2701102) and the National Nature Science Foundation of China (No. 62141222 and No. U20A20173).

REFERENCES

[1] "CVE-2017-5638," https://nvd.nist.gov/vuln/detail/CVE-2017-5638, 2018.

[2] "Equifax to pay at least $650 million in largest-ever data breach settlement," https://www.nytimes.com/2019/07/22/business/equifax-settlement.html, 2017, accessed: 2022-05-03.

[3] Wikipedia, "Coordinated vulnerability disclosure," https://en.wikipedia.org/wiki/Coordinated_vulnerability_disclosure, 2018.

[4] "Coordinated vulnerability disclosure policies in the eu," 2022. [Online]. Available: https://www.enisa.europa.eu/news/enisa-news/coordinated-vulnerability-disclosure-policies-in-the-eu

[5] "Asf project security for committers (apache.org)," 2022. [Online]. Available: https://www.apache.org/security/committers.html

[6] "About coordinated disclosure of security vulnerabilities - github docs," 2022. [Online]. Available: https://docs.github.com/en/code-security/repository-security-advisories/about-coordinated-disclosure-of-security-vulnerabilities

[7] "Microsoft's approach to coordinated vulnerability disclosure," 2022. [Online]. Available: https://www.microsoft.com/en-us/msrc/cvd

[8] "Project zero," 2022. [Online]. Available: https://googleprojectzero.blogspot.com/

[9] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2201–2215.

[10] 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 *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 705–716.

[11] "CVE-2021-44228," https://nvd.nist.gov/vuln/detail/CVE-2021-44228, 2021.

[12] "Restrict ldap access via jndi," https://github.com/apache/logging-log4j2/commit/755e2c9, 2022.

[13] "CWE view: Research concepts," http://cwe.mitre.org/data/definitions/1000.html, 2022.

[14] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *Proceedings of the 10th IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 2. IEEE, 2006, pp. 1735–1742.

[15] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *Proceedings of the 39th International conference on machine learning (ICML)*. PMLR, 2020, pp. 1597–1607.

[16] F. Tip, "A survey of program slicing techniques," *J. Program. Lang.*, vol. 3, 1995.

[17] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[18] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[19] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou *et al.*, "Mvp: Detecting vulnerabilities using patch-enhanced vulnerability signatures," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1165–1182.

[20] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μ vuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2019.

[21] "Our replication package," https://figshare.com/s/840e1fb94bd972829c80, 2022.

[22] T. Gao, X. Yao, and D. Chen, "Simcse: Simple contrastive learning of sentence embeddings," *arXiv preprint arXiv:2104.08821*, 2021.

[23] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," in *Proceedings of the 26th Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

[24] "NVD," https://nvd.nist.gov/, (Accessed on 04/08/2022).

[25] "Common vulnerability scoring system sig," https://www.first.org/cvss, (Accessed on 04/03/2022).

[26] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," *ACM Sigplan Notices*, vol. 19, no. 5, pp. 177–184, 1984.

[27] "tree-sitter: An incremental parsing system for programming tools," https://github.com/tree-sitter/tree-sitter, (Accessed on 04/11/2022).

[28] "Computes the ast difference between two spoon java source code abstract syntax trees," https://github.com/SpoonLabs/gumtree-spoon-ast-diff, (Accessed on 04/11/2022).

[29] Dejan Bosanac, "Fix xss in cron expressions," https://github.com/apache/activemq/commit/148ca81d, 2018.

[30] "CVE-2013-1879," https://nvd.nist.gov/vuln/detail/CVE-2013-1879, 2021.

[31] HAPI FHIR, "Fix a potential security vulneability in the testpage overlay," https://github.com/hapifhir/hapi-fhir/commit/8f41159e, 2018.

[32] "CVE-2019-12741," https://nvd.nist.gov/vuln/detail/CVE-2019-127418.

[33] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of EMNLP*, September 2020.

[34] "Vulnerability fix dataset — zenodo," https://zenodo.org/record/5513051/export/hx#.YnY7TdrMI2w, (Accessed on 04/12/2022).

[35] D. Spadini, M. Aniche, and A. Bacchelli, *PyDriller: Python Framework for Mining Software Repositories*, 2018.

[36] "NVD - data feeds," https://nvd.nist.gov/vuln/data-feeds, (Accessed on 04/11/2022).

[37] "NVD - vulnerability metrics," https://nvd.nist.gov/vuln-metrics/cvss.

[38] Y. Kim, "Convolutional neural networks for sentence classification," *The 19th 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 08 2014.

[39] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[40] M. Dong, Y. Li, X. Tang, J. Xu, S. Bi, and Y. Cai, "Variable convolution and pooling convolutional neural network for text sentiment classification," *IEEE Access*, vol. 8, pp. 16 174–16 186, 2020.

[41] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 810–822.

[42] 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.

[43] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE transactions on Signal Processing (TSP)*, vol. 45, no. 11, pp. 2673–2681, 1997.

[44] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems (TNNLS)*, vol. 28, no. 10, pp. 2222–2232, 2016.

[45] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *arXiv preprint arXiv:2011.14597*, 2020.

[46] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*, 2017.

[47] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2020, pp. 1433–1443.

[48] T. Zhang, B. Xu, F. Thung, S. A. Haryono, D. Lo, and L. Jiang, "Sentiment analysis for software engineering: How far can pre-trained transformer models go?" in *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 70–80.

[49] 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.

[50] C. Tantithamthavorn and A. E. Hassan, "An experience report on defect modelling in practice: Pitfalls and challenges," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. New York, NY, USA: Association for Computing Machinery, 2018, p. 286–295.

[51] 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 (TSE)*, vol. 34, no. 4, pp. 485–496, 2008.

[52] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan, "The impact of using regression models to build defect classifiers," in *Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 135–145.

[53] D. Romano and M. Pinzger, "Using source code metrics to predict change-prone java interfaces," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*. USA: IEEE Computer Society, 2011, p. 303–312.

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

[55] 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 *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 298–309.

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

[57] 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 (TSE)*, vol. 39, no. 6, pp. 757–773, 2012.

[58] 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 24th ACM SIGSOFT international symposium on foundations of software engineering (FSE)*, 2016, pp. 157–168.

[59] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering (TSE)*, vol. 34, no. 2, pp. 181–196, 2008.

[60] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE transactions on software engineering (TSE)*, vol. 33, no. 1, pp. 2–13, 2006.

[61] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 35th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 382–391.

[62] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 297–308.

[63] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 182–191.

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

[65] NVIDIA, "Nvidia v100," https://www.nvidia.com/en-us/data-center/v100, (Accessed on 04/15/2022).

[66] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426*, 2018.

[67] J.-R. Falleri, "Gumtreediff/gumtree wiki," https://github.com/GumTreeDiff/gumtree/wiki/Languages.

[68] Y. Ding, L. Buratti, S. Pujar, A. Morari, B. Ray, and S. Chakraborty, "Contrastive learning for source code with structural and functional properties," *arXiv preprint arXiv:2110.03868*, 2021.

[69] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, "Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation," *arXiv preprint arXiv:2108.04556*, 2021.

[70] X. Wang, Q. Wu, H. Zhang, C. Lyu, X. Jiang, Z. Zheng, L. Lyu, and S. Hu, "Heloc: Hierarchical contrastive learning of source code representation," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2022.

[71] N. D. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, 2021, pp. 511–521.

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