2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678720

Finding A Needle in a Haystack: Automated Mining of Silent Vulnerability Fixes

Jiayuan Zhou*, Michael Pacheco*, Zhiyuan Wan†, Xin Xia‡‖, David Lo§, Yuan Wang* and Ahmed E. Hassan¶
* Centre for Software Excellence, Huawei
† College of Computer Science and Technology, Zhejiang University, Hangzhou, China
‡ Faculty of Information Technology, Monash University, Melbourne, Australia
§ School of Information Systems, Singapore Management University, Singapore
¶ Software Analysis and Intelligence Lab (SAIL), Queen's University
{jiayuan.zhou1,michael.pacheco1,yuan.wang1}@huawei.com,wanzhiyuan@zju.edu.cn, Xin.Xia@monash.edu,
davidlo@smu.edu.sg, ahmed@cs.queensu.ca

***Abstract*—Following the coordinated vulnerability disclosure model, a vulnerability in open source software (OSS) is suggested to be fixed "silently", without disclosing the fix until the vulnerability is disclosed. Yet, it is crucial for OSS users to be aware of vulnerability fixes as early as possible, as once a vulnerability fix is pushed to the source code repository, a malicious party could probe for the corresponding vulnerability to exploit it. In practice, OSS users often rely on the vulnerability disclosure information from security advisories (e.g., National Vulnerability Database) to sense vulnerability fixes. However, the time between the availability of a vulnerability fix and its disclosure can vary from days to months, and in some cases, even years. Due to manpower constraints and the lack of expert knowledge, it is infeasible for OSS users to manually analyze all code changes for vulnerability fix detection. Therefore, it is essential to identify vulnerability fixes automatically and promptly. In a first-of-its-kind study, we propose VulFixMiner, a Transformer-based approach, capable of automatically extracting semantic meaning from commit-level code changes to identify silent vulnerability fixes. We construct our model using sampled commits from 204 projects, and evaluate using the full set of commits from 52 additional projects. The evaluation results show that VulFixMiner outperforms various state-of-the-art baselines in terms of AUC (i.e., 0.81 and 0.73 on Java and Python dataset, respectively) and two effort-aware performance metrics (i.e., *EffortCost*, P_{opt}). Especially, with an effort of inspecting 5% of total LOC, VulFixMiner can identify 49% of total vulnerability fixes. Additionally, with manual verification of sampled commits that were identified as vulnerability fixes, but not marked as such in our dataset, we observe that 35% (29 out of 82) of the commits are for fixing vulnerabilities, indicating VulFixMiner is also capable of identifying unreported vulnerability fixes.**

***Index Terms*—Software Security, Vulnerability Fix, Open Source Software, Deep Learning**

I. Introduction

Coordinated vulnerability disclosure (also known as responsible disclosure) [1], [2] is a widely used vulnerability disclosure model,[1,2,3] in which the relevant information of a vulnerability is only disclosed after a period of time that allows for the vulnerability to be fixed. Following this process, a vulnerability in an open source software (OSS) is suggested to be reported privately to the OSS maintainers, who "silently" fix the vulnerability (i.e., push the commit(s) to the source code repository, without explicit log messages indicating the vulnerability). They then integrate the fix "publicly" into the OSS (e.g., including the fix in a new release), and finally disclose the vulnerability and its fix.

It is important for OSS users to be aware of vulnerability fixes and apply fixes in time. In 2017, Equifax suffered from a data breach, compromising the personal information of over 143 million U.S. consumers, due to a missed security update [3], [4]. To be aware of vulnerability fixes, OSS users usually monitor the vulnerability disclosure information from public vulnerability advisories.

Common Vulnerabilities and Exposures (CVE) and the National Vulnerability Database (NVD) are two of the most popular public vulnerability advisories. NVD is a robust and widely used vulnerability advisory, fully synchronized with CVEs, and provides additional information (e.g., the severity) regarding vulnerabilities. Thus, it is a common practice for OSS users to monitor NVD primarily, as a method of becoming aware of vulnerabilities and their fixes. However, due to the slow progress in disclosing reported vulnerabilities [5], it is challenging to promptly discover vulnerability fixes. Also, the time interval between the availability of a vulnerability fix and its disclosure can vary from days to years[4], and the median of this time interval was reported to be more than one week in NVD [5]. As an example, CVE-2018-11776 [6] is a remote code execution vulnerability in Apache Struts, which is the same type as the critical vulnerability that led to the data breach of Equifax. This vulnerability was "silently" fixed in the public code repository in June 2018 [7], though the patch was disclosed two months later. Given the public nature of OSS development, once a vulnerability fix is pushed to the source code repository, a malicious party could infer the corresponding vulnerability and exploit it before the security patch is publicly integrated or disclosed. As a result, the users

‖Corresponding author.

[1]https://www.microsoft.com/en-us/msrc/cvd

[2]https://github.com/google/oss-vulnerability-guide

[3]https://www.apache.org/security/committers.html

[4]https://www.fireeye.com/blog/threat-research/2020/04/time-between-disclosure-patch-release-and-vulnerability-exploitation.html

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00068

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:46:17 UTC from IEEE Xplore. Restrictions apply.

of Apache Struts were exposed to huge security risks during the two months between patch availability and disclosure.

The disclosure latency will extend the vulnerability's window of exposure and it makes OSS users, especially the enterprise users who use OSS in their products, at a great disadvantage in defending security attacks. Hence, it is crucial to facilitate the awareness of vulnerability fixes for OSS users, so that they can react to the vulnerabilities as early as possible. For example, when enterprise users receive the notifications of vulnerability fixes, they can infer the corresponding vulnerability and evaluate the security impact on their product and take actions (e.g., recompile the software to incorporate the critical fixes that might cause huge loss) before the vulnerability is publicly disclosed. Without the early awareness, OSS users need to rush to address the vulnerability at the date of the disclosure. Hence, early awareness helps OSS users reduce the stress of keeping their system safe. In addition, being aware of the fix earlier could help shorten the vulnerability remediation time for OSS users. For example, with the help of hot-patching frameworks [8]–[11], hot-patches can be derived from the original vulnerability fixes to ease the deployment of security updates.

Following coordinated vulnerability disclosure in OSS contexts [1], [2], a vulnerability is suggested to be fixed silently, which means that any information indicating the nature of the vulnerability should not be exposed. For example, the vulnerability handling process of Apache[5] suggests messages associated with commits should not make any reference to any security-related nature. Hence, for general OSS users, the only way to be aware of silent vulnerability fixes promptly is to monitor and analyze the commit code changes constantly, which is infeasible due to manpower constraints. Therefore, it is essential to propose approaches that automatically identify silent vulnerability fixes in real-time.

Previous studies [12]–[15] leverage vulnerability-related artefacts (e.g., commit messages and issue reports) to identify vulnerability fixes. They extract features from textual data and leverage machine learning techniques to predict whether a commit is for fixing a vulnerability or not. Different from previous work, we incorporate a deep learning solution designed for analyzing the code of commits.

In our work, we propose VulFixMiner, a Transformer-based [16] model, to automatically identify silent vulnerability fixes in a practical setting (i.e., extremely imbalanced class distribution of the fixes). Since we are interested in silent vulnerability fixes, of which the commit message should not leak information related to the vulnerability, we only consider code change information. Given the outstanding ability in learning effective contextual representation, we leverage the Transformer-based [16] language model CodeBERT [17], which is pre-trained on a large programming language corpus, to learn the semantic meaning of code changes. We first fine-tune CodeBERT to learn the semantic meaning of file-level code changes, to generate contextual embedding vectors for changed files. The vectors are then aggregated to generate a commit-level contextual embedding vector, which is used for classification. VulFixMiner is capable of cross-project and cross-language just-in-time vulnerability fix identification.

We construct our model using 63,331 commits from 204 projects and evaluate it using 143,989 commits from 52 projects that are unseen during the training phase. The evaluation results show that VulFixMiner outperforms various state-of-the-art baselines in terms of AUC (i.e., 0.81 and 0.73 on Java and Python dataset, respectively), and two effort-aware evaluation metrics (i.e., *EffortCost*, P_{opt}). Particularly, with an effort of inspecting 5% of the total LOC, VulFixMiner can identify 49% of total vulnerability fixes. Additionally, by manually investigating a sample of commits identified as vulnerability fixes that are not marked as vulnerability fixes in our dataset, we observe 35% (29 out of 82) of them are for fixing vulnerabilities, indicating VulFixMiner is also capable of identifying unreported vulnerability fixes.

In summary, this paper makes the following contributions:

- We propose the use of VulFixMiner to automatically identify silent vulnerability fixes, supporting both cross-project and cross-language scenarios.
- To the best of our knowledge, we are the first to use deep learning techniques to identify silent vulnerability fixes in a practical setting.
- VulFixMiner achieves higher discriminative power and efficiency compared to state-of-the-art baselines in identifying silent vulnerability fixes, when evaluated on a complete set of commits from 52 Java and Python projects.
- We find that VulFixMiner is capable of identifying unreported vulnerability fixes.
- To promote future work, we release the vulnerability fixes in our study [18].[6]

Paper organization. This paper is organized as follows: Section II introduces background information. We elaborate on our approach on Section III. Section IV presents the experimental procedures and results. In Section V, we discuss the unreported vulnerability fixes, the ethical consideration, the other application scenarios, the commit messages of silent vulnerability fixes, and time efficiency. Sections VI and VII cover the possible threats to validity, and highlight related work, respectively. We conclude this paper and discuss future work opportunities in Section VIII.

II. Preliminaries

In this section, we briefly introduce the vulnerability disclosure models, Common Vulnerabilities and Exposures, National Vulnerability Database, and pre-trained NLP models for natural language tasks.

A. Vulnerability disclosure models

Full Disclosure. In the Full Disclosure model [19], the vulnerability is fully disclosed onto public disclosure channels as early as possible. The idea behind this model is that the early

[5]https://www.apache.org/security/committers.html

[6]Note that we are undergoing the company's internal process for model publishing.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:46:17 UTC from IEEE Xplore. Restrictions apply.

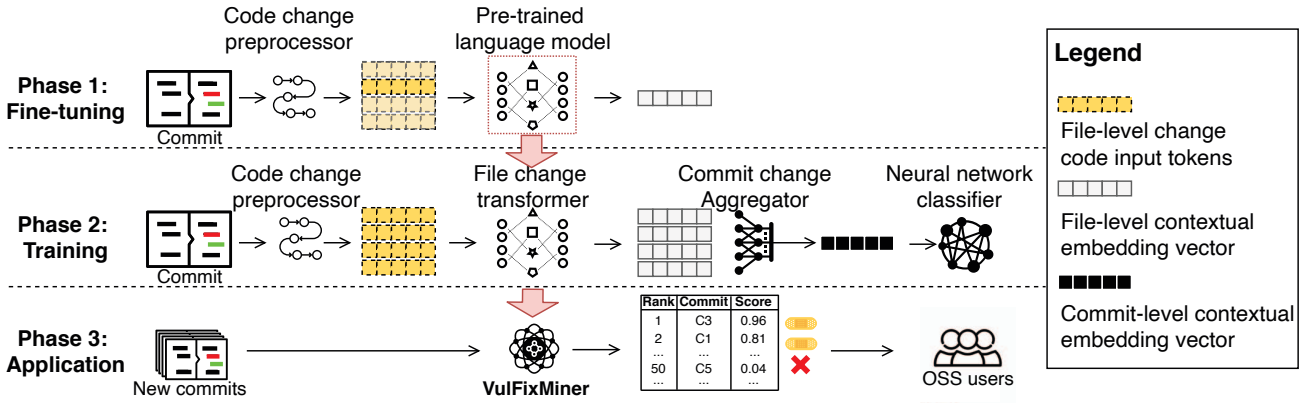

Fig. 1: The overall framework of VulFixMiner.

public awareness of the vulnerability will benefit users who are vulnerable, more than malicious actors.

Coordinated Vulnerability Disclosure. In the Coordinated Vulnerability Disclosure model [1], [2], a vulnerability remains undisclosed, or silent, as long as possible to provide developers of the software enough time to fix it. After a new release including the fix is published, the vulnerability is disclosed. This model is commonly and widely applied in many OSS organizations[7,8,9]. In this disclosure model, users remain at risk as long as they are unaware that they are vulnerable. Also, a fix for the vulnerability might be committed to an OSS repository before the vulnerability is disclosed, which we refer to as a "silent fix". Due to the lack of public exposure of the vulnerability, it is expected that exploitation from malicious users will be less likely. However, given the public nature of OSS development, if a vulnerability is silently fixed before the fix is integrated, a malicious party can discover these fixes from the public code repository, and exploit the vulnerability in the vulnerable systems.

Non-disclosure. In the non-disclosure model [19], no details of a vulnerability, nor its fix, are disclosed. The software developers, users, and malicious attackers do not have access to any public information regarding these vulnerabilities. The risk involved with this model appears as users are never notified of vulnerabilities within the software. Although the non-disclosure model prevents attackers from analyzing security fixes and stops them from further finding similar vulnerabilities,[10] OSS users might not be aware of the urgency of a security fix and do not apply the fix. In this case, OSS users are always exposed to security risks. In our paper, we refer to these fixes as "unreported vulnerability fixes". We discuss the capability of VulFixMiner in identifying such fixes in Section V-A.

[7]https://www.microsoft.com/en-us/msrc/cvd
[8]https://googleprojectzero.blogspot.com/
[9]https://www.apache.org/security/committers.html
[10]https://redmondmag.com/articles/2011/02/16/microsoft-silent-fix-due-diligence.aspx

B. Common Vulnerabilities and Exposure (CVE), and the National Vulnerability Database (NVD)

CVE, developed by The Mitre Corporation (MITRE), provides a standardized method for software developers to disclose, identify, and manage software vulnerabilities. Once a vulnerability is identified, the OSS developers can request a unique CVE ID from a CVE Numbering Authority (CNA), such as MITRE, for the vulnerability. Once this ID is assigned, the CVE will be included in a vulnerability database for public disclosure. CVE is one of the most popular vulnerability advisories; OSS users monitor them to discover vulnerabilities that might impact them, and the fixes to defend against exploitation.

NVD[11] is a popular CVE database, maintained by the National Institute for Science and Technology (NIST). NVD is automatically synchronized with the MITRE database and includes all CVE information included in MITRE, as well as additional information. More specifically, NVD assigns labels to each URL in the list of references of a CVE, including the vulnerability fix. This labeling can only be performed if the vulnerability fix is already disclosed, MITRE or NVD become aware of it, and MITRE or NVD have included it in their databases.

C. Pre-Trained NLP models for natural language tasks.

Transformer. The Transformer [16] is a model originally designed for tasks related to natural language processing (NLP). The canonical architecture of the model consists of an encoder-decoder and includes several layers with attention mechanisms. This architecture allows for the use of pre-trained models, which have already been trained on related domain-specific data by others, and can be reused for multiple related tasks through fine-tuning.

CodeBERT. CodeBERT [17], a variant of RoBERTa [20], is a multi-layer bidirectional Transformer, and one of the first models to incorporate both natural language (NL) and programming language (PL) information. As there is a natural difference between NL and PL, CodeBERT is further pre-trained on RoBERTa with 2.1 million function-level source

[11]https://nvd.nist.gov/

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:46:17 UTC from IEEE Xplore. Restrictions apply.

code data, together with the corresponding function documentation. This allows CodeBERT to handle NL to PL-related tasks, including NL code search, NL-PL probing, and code documentation generation.

III. Proposed Approach

In this section, we first introduce the overall framework of VulFixMiner, then we describe the details of each module in VulFixMiner.

A. Overall Framework

The goal of VulFixMiner is to effectively identify vulnerability fixes using code change information. Figure 1 shows the overall framework of VulFixMiner, which consists of three phases (i.e., Fine-tuning, Training, and Application). We first fine-tune a pre-trained language model to learn the representation of file-level code change in the Fine-tuning Phase. In the Training Phase, we consider the fine-tuned model as the *file change transformer*, and we use the *commit change aggregator* to aggregate the file-level code change representations into commit-level code change representations, and then train a neural network classifier to identify commits. In the Application Phase, the trained VulFixMiner consumes new commits from OSS repositories and computes scores, which indicate the likelihood that a commit is for fixing vulnerabilities. We elaborate on the details of each phase as follows:

Fig. 2: A sample commit which is for fixing CVE-2018-11776 vulnerability [6].

B. Phase 1: Fine-tuning

We choose CodeBERT [17] as our pre-trained language model. We first consider code changes as a sequence of tokens in a BoW. The code change preprocessor (See Figure 1) extracts code changes from commits and constructs a file-level sequence of input tokens, which are then fed into CodeBERT for fine-tuning. We introduce the code change preprocessing and CodeBERT fine-tuning as below:

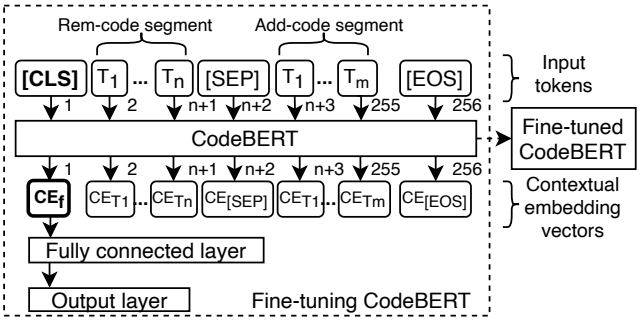

Fig. 3: The architecture of fine-tuning CodeBERT. CE_f is the embedding vector of the classification token *[CLS]* and the vector works as the contextual embedding of the input tokens.

1) Code change preprocessing: As shown in Figure 2, a commit may contain code changes across multiple files. We preprocess code changes in the following three steps:

1. Extract file-level code changes. We first extract the information of file-level changes into separate code change documents. For each document, we further extract removed and added code lines and split them into the *rem-code* and *add-code* segments, respectively.

2. Process the removed code and added code lines. For each segment, we first tokenize code lines into a sequence of tokens, and then further split camel case style (e.g., "*addData*") and snake case style (e.g., "*add_data*") tokens, by using *codeprep* [21].

3. Input tokens construction. To construct the same input representation in CodeBERT [17], we use the same tokenizer[12] that CodeBERT used to concatenate two segments. Three separator tokens (i.e., *[CLS]*, *[SEP]*, and *[EOS]*), are used for concatenation. *[CLS]* is a special classification token found always as the first token of input. The final hidden state of CodeBERT corresponding to *[CLS]* is considered as the contextual embedding (i.e., aggregated sequence embedding) of the input tokens, and can be used for classification. *[SEP]* and *[EOS]* are used to separate two segments, and to indicate the end of input, respectively. All inputs are either padded or truncated to the same length (i.e., 256) by the CodeBERT tokenizer. Figure 3 shows an example of the constructed input tokens.

2) CodeBERT fine tuning: The downstream fine-tuning task is to predict the probability that the code changes in a file are for fixing vulnerabilities. Figure 3 shows the architecture of fine-tuning CodeBERT. Note that we use the same CodeBERT tokenizer to generate the input embedding vector. Since it is not the main focus of our approach, we omit the details of the input embedding generation (which is explained in the original work of CodeBERT [17]). During fine-tuning, the output of CodeBERT includes the contextual embedding vectors (i.e., CE_T) of each token and an embedding vector (i.e., CE_f) of *[CLS]*, which works as the contextual embedding of the input tokens. We use the contextual embedding vector of the

[12]https://huggingface.co/microsoft/codebert-base/tree/main

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:46:17 UTC from IEEE Xplore. Restrictions apply.

input tokens (i.e., the removed and the added code tokens) to represent the semantic relevance between removed and added code lines. Then we connect CE_f with a fully connected layer followed by an output layer, which computes a score indicating the probability that the code changes in a file are for fixing vulnerabilities. To improve the robustness of our model, we apply the dropout technique [22] on the fully connected layer.

During fine-tuning, the parameters of CodeBERT are learned to minimize the cross-entropy loss. After fine-tuning, all parameters of CodeBERT are frozen, and the fine-tuned CodeBERT is used as a file change transformer to consume file-level input tokens, and output contextual embedding vectors of the file-level code changes.

C. Phase 2: Training

Following the same code change preprocessing in Section III-B1, the constructed file-level input tokens are encoded into file-level contextual embedding vectors (i.e., CE_f) by the file change transformer, which is the fine-tuned CodeBERT. We then use a commit change aggregator to aggregate the file-level vectors belonging to the same commit, to generate a unified file-level code change representation (i.e., CE_{commit}), representing commit-level code changes.

In the commit change aggregator, we compute the element-wise mean for the vectors of files belonging to the same commit as $CE_{commit} = \frac{\sum_{i=1}^{n} CE_{fi}}{n}$, where CE_{fi} is the contextual embedding vector of the i[th] file in a given commit, and n is the number of files belonging to that commit.

Finally, we use a one-layer neural network classifier to classify commits. The aggregated CE_{commit} is fed into a fully connected layer, followed by an output layer which computes a score indicating the probability that the code changes in a commit are for fixing vulnerabilities. We also apply the dropout technique [22] on the fully connected layer here.

D. Phase 3: Application

After the Training Phase, OSS users can use the trained VulFixMiner to mine vulnerability fixes from new commits. For example, VulFixMiner can be integrated into an automatic monitoring service of OSS code repositories. Given a set of new commits, VulFixMiner computes scores for each of them and outputs a commit ranking to OSS users. The commits are ranked by scores and the higher score of a commit indicating the higher probability of the commit being for fixing vulnerabilities (see Figure 1).

IV. Experiments

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

- **RQ1: How effective is VulFixMiner compared to the state-of-the-art baselines?**
- **RQ2: Does VulFixMiner benefit from fine tuning using cross-domain data?**

Next, we describe our dataset and each step of data processing. We then present the details of five baselines, four evaluation metrics, and experiment setup. Finally, we show our research questions and results.

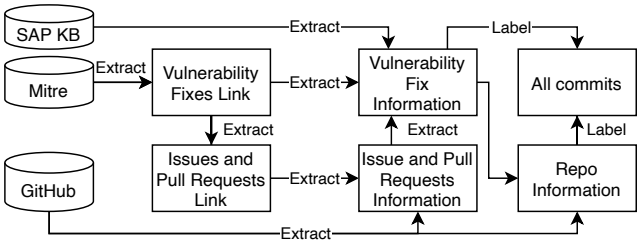

Fig. 4: An overview of our data collection approach. We collect vulnerability data from *SAP KB* project [23] and Mitre. We collect commit data from GitHub.

A. Data Collection

Our dataset consists of commit information from a set of Java and Python OSS. We rely on two vulnerability-related data sources to first collect a set of commits that are for fixing vulnerabilities and then collect all commits of OSS that contain any of the vulnerability fixes we collected. Figure 4 illustrates the overview of data collection approach.

1. Collecting vulnerability fix links. We first collect data from an existing Java vulnerability fix dataset [23], which is manually curated by *SAP KB* project.[13] From this data source, we obtain 1,055 vulnerability fix links, spanning across 183 Java OSS projects and corresponding to 615 CVEs.

Next, we collect all CVEs (disclosed by January 26, 2021) from Mitre CVE database[14], resulting in 201,234 CVEs. We proceed by extracting CVEs containing a patch fixing that vulnerability, with a link toward a GitHub[15] commit, issue, or pull request. We then collect the commit information and filter for commits that contain Java or Python file extensions. For Java, we obtain a total of 199 commits, 227 issues, and 155 pull requests, spanning across 189 projects and corresponding to 340 CVEs. For Python, we obtain a total of 288 commits, 244 issues, and 353 pull requests, spanning across 256 projects and corresponding to 444 CVEs. We then merge all commits collected up until this point and remove any duplicates. We note that in these steps we include vulnerability fixes regardless of when they were disclosed in relation to the vulnerability itself.

2. Collecting issue and pull request information. We proceed by collecting the commit links related to the total number of issues (471) and pull requests (508) from Java and Python projects using GitHub. This results in 383 Java vulnerability fixes, spanning across 101 projects and corresponding to 186 CVEs. As for Python, this results in 597 vulnerability fixing commits spanning across 141 projects and corresponding to 233 CVEs. We add this result to the list from the previous step and remove any duplicates.

3. Collecting OSS repository and commit information. Finally, we collect the commit information from each vulnerability fix link and collect all commits made within their

[13]https://sap.github.io/project-kb/
[14]https://cve.mitre.org/
[15]https://github.com/

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:46:17 UTC from IEEE Xplore. Restrictions apply.

projects, up until February 26, 2021. If their IDs are found within the list of vulnerability fixes from the previous step, we label these commits by assigning a "1" to them, defining the positive label.

As a result, the Java dataset includes 1,436 vulnerability fixes and 839,682 non-vulnerability fixing commits, spanning across 310 projects and corresponding to 839 CVEs. The Python dataset includes 885 vulnerability fixes and 722,291 non-vulnerability fixing commits, spanning 256 projects and corresponding to 444 CVEs.

B. Data Preprocessing

We further preprocess our dataset to filter out noisy data and enhance existing data.

1. Remove noisy data. First, we remove duplicate code change information, keeping the first instance. Next, we remove OSS that only have one vulnerability fix. One challenge for identifying vulnerability fixes in a practical setting is the extremely imbalanced dataset. We expect that it is likely for these specific OSS to offer less opportunity for our models to learn and differentiate between vulnerability and non-vulnerability fixing commits. We then remove large commits that are less likely to fix vulnerabilities, by calculating two thresholds using the 95th percentile of the total modified lines of code and the number of changed files of vulnerability fixes. For Java, these values are 309 and 15 respectively. The removal results in 474,555 non-vulnerability fixing commits, and 1,353 vulnerability fixes, across 150 projects. For Python, these values are 80 and 6 respectively. The removal results in 357,696 non-vulnerability fixing commits, and 751 vulnerability fixes, across 106 projects.

2. Enhance the dataset by identifying more secret vulnerability fixes. Next, we enhance our dataset by labeling more commits that are relevant to vulnerability fixes. To do this, we define a regular expression adapted from the work of Zhou and Sharma [12]. The original regular expression contains a diverse set of security-related keywords, and as a result, commonly mislabels commits. For example, the message of a commit[16] is "Migrate xml-insecure groovy −> java", containing a security-related keyword, "insecure", though the purpose of the commit is for code refactoring. To avoid such false alarms, we select three conservative keywords (i.e., "vuln", "CVE", and "NVD") from [12] to generate a new regular expression. We use this regular expression to relabel commits not identified by analyzing the fix links (described in Section IV-A) as vulnerability fixes, by matching the words found within their commit messages. In the Java dataset, we relabel 420 non-vulnerability commits across 123 OSS. In the Python dataset, we relabel 501 non-vulnerability commits across 98 OSS.

3. Split dataset into training, testing, and validation sets. In this step, we first split the data project-wise, using an 80%/20% split, and consider the 20% split as the testing dataset. We then further split the 80% training set project-wise, into a 90%/10% split, and consider the 10% split as the validation dataset, and the remaining split as the training dataset. In this case, the commits in test data are from different projects which are never seen in the training and validation sets.

We randomly undersample non-vulnerability fixing commits for each OSS, and combine the result with the unmodified vulnerability fixes, to form our final training and validation datasets. We perform this approach as another mechanism to reduce the imbalanced nature of the dataset; however, we note that these resulting datasets continue to remain imbalanced. Also, we do not apply this undersampling to the test dataset, as our goal is to evaluate our model in a practical setting.

Table I describes our final dataset. We notice an extremely imbalanced class distribution of vulnerability fixes. For the percentage of vulnerability fix within each OSS, the median number is 0.35%. The median number of vulnerability fixes within each OSS is only 4.

C. Baselines

We compare VulFixMiner with several baselines:

- *RandomGuess:* Random Guess is a strawman baseline that randomly predicts whether a code change is for fixing vulnerabilities.
- *SVM:* The closest work to ours is by Sabetta and Bezzi [13]. Their work treated code and commit messages as a collection of tokens in a bag of words (BoW), and trained two linear support vector machine (SVM) models basing on commit messages and code of commit, respectively. We replicate the code-based model.
- *RandomForest:* It is widely applied throughout software engineering studies (e.g., defect prediction [24] and security bug report prediction [25]) and showed robust and high performances [26]–[28].
- *LSTM:* Long short-term memory (LSTM) network [29] is a widely applied RNN-based model throughout software engineering studies [30]. Instead of treating code as a bag of words, we use LSTM to represent the sequential information of code by considering code as a sequence of tokens.
- *Transformer*: Transformer [16] is an encoder-decoder, neural network-based language model with multi-head attention layers, giving way to the use of pre-trained models (e.g., CodeBERT), and is used for a variety of natural language processing tasks.

D. Evaluation Metrics

Given the fact that the vulnerability fixes are rare among the code commits (i.e., the extremely imbalanced dataset scenario), the goal of VulFixMiner is to help users reduce the efforts in identifying vulnerability fixes from a large scope of code commits. We leverage two effort-aware performance metrics (i.e., *CostEffort@L* and P_{opt} [31]–[34]) to evaluate the effectiveness of VulFixMiner in practical setting. Similar to prior studies [35]–[38], instead of using threshold-dependent measures (e.g., precision and recall) which depend on arbitrarily thresholds and are sensitive to imbalanced data, we use

[16]https://github.com/spring-projects/spring-security/commit/2e2b22f87ecbd40e3328a089efb3189a3b9cdd99

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:46:17 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Description of our dataset after preprocessing. We refer to vulnerability fixes and non-vulnerability fixing commits as V.F. and N.V.F, respectively.

	Training Set			Validation Set			Testing Set		
	#V.F.	#N.V.F	#Projects	#V.F.	#N.V.F	#Projects	#V.F.	#N.V.F	#Projects
Java	983	31,323	120	191	6,921	119	300	87,856	30
Python	522	20,362	84	80	2,949	83	195	55,638	22

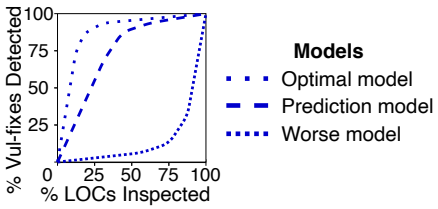

Fig. 5: An example of the relationship between the percentage of vulnerability fixes detected and the amount of inspection cost (i.e., %LOC) for different prediction models.

AUC to quantify the discriminative capability of VulFixMiner. We introduce each measure as follows:

CostEffort@L: We wish to identify more vulnerability fixes under the limited inspection effort. Similar to the effort evaluation in the defect prediction task [31]–[33], [39], we also consider the LOC as the proxy of the inspection effort. *CostEffort@L* is the proportion of inspected vulnerability fixes among all the actual vulnerability fixes when L LOC of all commits are inspected. *CostEffort@L* is computed as $\frac{n}{N}$, where n accounts for the correctly identified vulnerability fixes, N accounts for the total vulnerability fixes, and L accounts for $L\%$ of total LOC of commits. The high *CostEffort@L* indicates more vulnerability fixes could be identified costing the effort of inspecting L LOC. We calculate *CostEffort@5%* and *CostEffort@20%* in our study.

P_{opt}*:* is the normalized version of the effort-aware performance metrics which is first introduced by Mende and Koschke [34], basing on the concept of the "code-churn-based" Alberg diagram [40]. P_{opt} is a widely used effort-aware performance metric in defect prediction [31]–[33], [39]. In our study, we calculate P_{opt} as same as they do.

Figure 5 is an example of the Alberg diagram in the vulnerability fix detection context, where the diagram shows the relationship between the percentage of identified vulnerability fixes (i.e., y-axis) achieved by a model and the percentage of LOCs that are inspected (i.e., x-axis). The optimal and worst model represents the cases where all commits are respectively sorted in decreasing and ascending order by *vul-fix-density*. The *vul-fix-density* of a commit c is defined as $D(c) = \frac{Y(c)}{Effort(c)}$, where *Y(c)* is 1 if the commit c is vul-fix and 0 otherwise, and *Effort(c)* is the LOC the commit c. By doing this, in Figure 5 the points on the optimal model and worst model represent the maximum and minimum percentage of vul-fix detected, respectively, with %LOCs inspected. Then we can use optimal model and worst model as the upper bound and lower bond, respectively, to further assess the prediction model.

For a given prediction model m, its P_{opt} is computed as $P_{opt}(m) = \frac{Area(O,P)}{Area(O,W)}$, where O, P, and R represent the optimal model curve, the prediction model curve, and the worst model curve, respectively, and the function $Area(curve1, curve2)$ represents the corresponding area between the two curves. A larger P_{opt} value indicates a smaller difference of performance between the prediction model and the optimal case. In our study, we calculate P_{opt} when 5% and 20% of the LOCs are inspected and we denote them as P_{opt}@5 and P_{opt}@20, respectively.

AUC: is the area under the receiver operating characteristic (ROC) curve [41], which measures the prediction performance of the model for all possible classification thresholds (i.e., from 0 to 1). It is robust in quantifying the discriminative capability of a classifier, especially in imbalanced class distributions, due to its insensitivity toward them [26]. The *AUC* has been recommended as the primary metric to determine and compare the performance of classifiers [42], and should be used over other metrics which are threshold-dependent (e.g., F1-score) [43]. A classifier with an $AUC \geq 0.7$ ($0 \leq AUC \leq 1$) is considered to have achieved an acceptable performance [44]. In our experiment, the *AUC* measures the probability that our classifier will rank a randomly selected vulnerability fixing commit higher than a randomly selected non-vulnerability fixing commit.

E. Experiment Setup

For VulFixMiner, we use the CodeBERT tokenizer for input embedding vector generation. The size of the vocabulary is 50,265 and the size of the input embedding vector is 256. The sizes of the fully connected layers described in Sections III-B2 and III-C are both set to 768, which is the same size as the hidden state of CodeBERT. Before computing the score, a dropout [22] rate of 0.1 is used for both fully connected layers in the fine-tuning and classification steps.

VulFixMiner is fine-tuned using Adam [45] with shuffled mini-batches. During fine-tuning, we set the learning rate of Adam to 1e-5 and the batch size to 8. We fine-tune CodeBERT for 15 epochs with an early stopping strategy [46], [47] to avoid overfitting problems during the fine-tuning process. We stopped the fine-tuning if the value of the cross-entropy loss has not been updated on the validation dataset in the last 5 epochs. We train the neural network classifier with a batch size of 32 for 60 epochs with an early stopping strategy.

The input of each baseline model is the file-level code change tokens. Except for adding separator tokens and applying truncating and padding, we process the code changes in the same way as what we do in Section III-B1. We choose

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:46:17 UTC from IEEE Xplore. Restrictions apply.

4,000 most frequent tokens in the training set to build the token vocabulary.

For the SVM baseline, we use the same hyperparameter setting in [13]. Similar to [48], for the RandomForest baseline, we tune the $mtry$ (i.e., number of randomly sampled variables). We find that using the square root of the feature number (i.e., the size of vocabulary) can achieve the best performance on the validation dataset. Hence, we set $mtry$ to 200. LSTM baseline has a single-layer LSTM network with an unrolling length of 32 and a hidden unit size of 256. For transformer baseline, except for the size of hidden states, we use the same hyperparameter setting in [16]. We set the size of hidden states to 768, which is the same size used in CodeBERT.

Our experiments are conducted on EulerOS v1.13.10-r1 64 bits, with a V100-32GB GPU.[17].

F. Research Questions and Results

RQ1: VulFixMiner vs. Baselines

To answer this RQ, we evaluate VulFixMiner and baseline models on Java and Python test dataset respectively, in terms of *AUC*, *CostEffort@5%*, *CostEffort@20%*, P_{opt}@5%, and P_{opt}@20%. Tables II and III show the evaluation results on Java and Python projects, respectively. We observe that VulFixMiner outperforms all baselines on both Java and Python projects in terms of all evaluation performance metrics.

For Java projects, VulFixMiner achieves *AUC*, *CostEffort@5%*, *CostEffort@20%*, P_{opt}@5%, and P_{opt}@20% of 0.81, 0.61, 0.71, 0.53, and 0.63, respectively. Using RandomForest, the best performing model for comparison, these results constitute improvements of 1%, 41%, 14%, 39%, and 24%, respectively. VulFixMiner thus achieves about the same AUC as RandomForest, but outperforms by substantial margins in terms of *CostEffort@5%* and P_{opt}@5%, which indicates VulFixMiner can reduce the inspection effort in a practical use. When comparing to RandomGuess, SVM, LSTM, and Transformer, VulFixMinerava outperforms them in terms of all evaluation metrics by large margins.

For Python projects, VulFixMiner achieves *AUC*, *CostEffort@5%*, *CostEffort@20%*, P_{opt}@5%, and P_{opt}@20% of 0.73, 0.32, 0.56, 0.24, and 0.39, respectively. Using all baselines for comparison, these results constitute improvements of 11-23%, 18-26%, 18-41%, 16-21%, and 18-30% respectively.

These results indicate that VulFixMiner has high discriminative power in identifying vulnerability fixes, and is capable of identifying vulnerability fixes with less inspection effort in practical use.

RQ2: Impact of Cross-domain Data

To answer this RQ, we construct three variants of VulFixMiner with different fine-tuning strategies: 1) without fine-tuning, 2) only fine-tuning with Java projects, 3) only fine-tuning with Python projects, and we refer these three variants as to VulFixMiner$_{NoFT}$, VulFixMiner$_J$, and VulFixMiner$_P$, respectively. We evaluate these three variants using the same evaluation metrics on Java and Python projects. Tables IV and V show the evaluation results on Java and Python projects, respectively. We observe that VulFixMiner$_{NoFT}$ performs the worst on both tasks. Comparing VulFixMiner to VulFixMiner$_{NoFT}$, we find that the fine-tuning process improves VulFixMiner in terms of *AUC*, *CostEffort@5%*, *CostEffort@20%*, P_{opt}@5%, and P_{opt}@20% by 14-17%, 27-50%, 42-44%, 22-46%, and 31-42%, respectively. We also observe that VulFixMiner$_J$ and VulFixMiner$_P$ outperform all baselines on Java and Python projects, respectively. However, the performance of VulFixMiner, which is fine tuned with both Java and Python data, outperforms both VulFixMiner$_J$ and VulFixMiner$_P$ on both Java and Python projects

When fine-tuning is done using data from only one domain (e.g., Java projects), the fine-tuned model achieves better performance on the task within the same domain. For example, VulFixMiner$_J$ is fine-tuned using Java projects, and performs better than VulFixMiner$_P$ on Java projects. One possible explanation is different programming languages vary in their syntactical features. Hence, the knowledge learned solely from one language may not be aligned well on the task involving code in another language.

However, the knowledge learned from one domain may contribute to the task belonging to the other domain. Figure 6 shows Venn diagrams indicating the number of vulnerability fixes identified by VulFixMiner$_J$ and VulFixMiner$_P$, with an inspection effort of 5% of the total LOC. In Java projects, we observe that VulFixMiner$_P$ identifies 11% (32 out of 300) Java vulnerability fixes, which cannot be identified by VulFixMiner$_J$. On Python projects, VulFixMiner$_J$ identifies 14% (27 out of 195) Python vulnerability fixes, which cannot be identified by VulFixMiner$_P$. This could be a possible explanation of why VulFixMiner benefits from fine-tuning with cross-domain data.

The overall performance of VulFixMiner on both Java and

TABLE II: Performance of VulFixMiner and baseline models for the Java projects.

Model	AUC	CostEffort		P_{opt}	
		@5%	@20%	@5%	@20%
RandomGuess	0.50	0.07	0.19	0.05	0.20
SVM	0.59	0.20	0.57	0.14	0.39
RandomForest	0.80	0.44	0.6	0.41	0.50
LTSM	0.52	0.05	0.22	0.03	0.11
Transformer	0.64	0.33	0.50	0.06	0.24
VulFixMiner	**0.81**	**0.61**	**0.71**	**0.53**	**0.63**

TABLE III: Performance of VulFixMiner and baseline models for the Python projects.

Model	AUC	CostEffort		P_{opt}	
		@5%	@20%	@5%	@20%
RandomGuess	0.53	0.06	0.23	0.05	0.20
SVM	0.55	0.14	0.15	0.08	0.14
RandomForest	0.62	0.11	0.31	0.08	0.18
LTSM	0.50	0.06	0.18	0.03	0.09
Transformer	0.56	0.08	0.38	0.04	0.21
VulFixMiner	**0.73**	**0.32**	**0.56**	**0.24**	**0.39**

[17]https://www.nvidia.com/en-us/data-center/v100/

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:46:17 UTC from IEEE Xplore. Restrictions apply.

TABLE IV: Performance of VulFixMiner and 3 variants on Java projects.

	AUC	CostEffort		P_{opt}	
		@5%	@20%	@5%	@20%
VulFixMiner$_{NoFT}$	0.64	0.11	0.37	0.07	0.21
VulFixMiner$_J$	0.8	0.52	0.66	0.46	0.57
VulFixMiner$_P$	0.61	0.24	0.42	0.17	0.3
VulFixMiner 0	**0.81**	**0.61**	**0.71**	**0.53**	**0.63**

TABLE V: Performance of VulFixMiner and 3 variants on Python projects.

	AUC	CostEffort		P_{opt}	
		@5%	@20%	@5%	@20%
VulFixMiner$_{NoFT}$	0.59	0.05	0.14	0.02	0.08
VulFixMiner$_J$	0.71	0.26	0.49	0.18	0.33
VulFixMiner$_P$	0.66	0.3	0.53	0.21	0.38
VulFixMiner	**0.73**	**0.32**	**0.56**	**0.24**	**0.39**

Python projects are 0.77, 0.49, 0.62, 0.41, and 0.53 for *AUC*, *CostEffort@5%*, *CostEffort@20%*, P_{opt}@5%, and P_{opt}@20%, respectively.

V. Qualitative Analysis and Discussion

A. Identification of Unreported Vulnerability Fixes

During our study, we observed that not all of the vulnerabilities and the corresponding fixes have been reported to the CVE database. Thus, we conduct a user study to evaluate the usefulness of our approach for identifying those secret (i.e., unreported) vulnerability fixes.

Experimental Tasks. We create tasks based on the 577 commits that are identified as false positives by VulFixMiner (with 0.5 as the threshold). Specifically, we randomly selected a statistically representative sample of 82 commits from these false positives (with a 95% confidence level and 10% confidence interval), which belong to 20 OSS systems. Each OSS system has 1 to 27 commits in the sampled set. For each commit, we ask two questions:

- Q1. Does this commit fix a security vulnerability?
- Q2. If the answer to Q1 is "Yes", what type of vulnerability does the commit fix?

In terms of the second question, we provide 11 options for types of vulnerabilities as referring to the top 10 frequent CWE software vulnerabilities [5], including (1) Buffer Overflow, (2) Improper Input Validation, (3) Access Control Error, (4) Cross-Site Scripting, (5) Information Disclosure, (6) Numeric Error, (7) Resource Management Error, (8) Race Condition, (9) SQL Injection, (10) Cryptographic Issues, and (11) Other.

Participants. We invite 6 security experts who have 5 to 12 years of experience with software security to participate in our user study. We ask each of them to finish an experimental task that includes 12 to 15 commits.

Results. Among the 82 commits, 29 commits are confirmed by the security experts as fixes of security vulnerabilities. The result indicates that our approach can effectively identify secret vulnerability fixes. In addition, the security experts suggest that

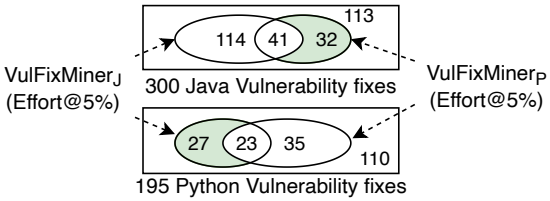

Fig. 6: Venn diagrams showing the number of identified vulnerability fixes by VulFixMiner$_J$ and VulFixMiner$_P$ with inspection effort of 5% of total LOC.

TABLE VI: An example commit that fixes an unreported vulnerability related to XXE and XML bomb attack.

[Jul 20, 2017] Key Code Change Snippet:

```
- private static final XMLInputFactory inputFactory
      = XMLInputFactory.newInstance();
+ private static final XMLInputFactory inputFactory
      = StaxUtils.createDefensiveInputFactory();
```

[Jul 20, 2017] Commit Message: XmlEventDecoder uses common defensive XMLInputFactory (now in StaxUtils). Issue: SPR-15797.

[Jul 20, 2017] Related Issue Title: [SPR-15797] Disable DTD and external entities support in XmlEventDecoder to prevent XXE and XML bomb attack.

[Sep 22, 2017] Commit Comment: @dmak: "... In our application we are also concerned about the security and would like to re-use the code from StaxUtils.createDefensiveInputFactory(). ..."

the 29 fixed security vulnerabilities can be further classified into 6 categories, i.e., 12 of resource management error, 10 of improper input validation, 4 of access control error, 1 of information exposure, 1 of cross-site scripting, and 1 null pointer dereference. Table VI shows an example commit[18] that fixes a resource management error vulnerability in the spring-framework project on Jul 20, 2017. In the code snippet of the example commit, the developer replaces "XMLInputFactory" with a defensive "XMLInputFactory" variant. Interestingly, the corresponding issue report confirms that the commit can prevent XXE and XML bomb attacks by disabling the support of DTD and external entities in "XMLEventDecoder". Also, another contributor of the spring-framework project came across this commit on September 21, 2017, commenting that he would like to port this commit to another branch to improve security.

B. Ethical consideration

VulFixMiner identifies silent vulnerability fixes before these fixes are disclosed. Although attackers might use the pre-disclosed information to gain temporal advantages, OSS users can benefit from our tool by being aware of the fixes as soon as they are developed, which is usually one week earlier than the disclosure [5]. Hence, we are not "arming" the potential attackers. Instead, we help OSS users become aware of the vulnerability earlier and point out the corresponding fix, so they could have more time and ease in defending against potential attacks.

[18]https://github.com/spring-projects/spring-framework/commit/e4651d6b50c5bc85c84ff537859c212ac4e33434

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:46:17 UTC from IEEE Xplore. Restrictions apply.

TABLE VII: The number of commit messages that are security related and not security related according to the regular expression that is used to match security-related issues [12].

Commit type	#Commit messages	
	Security related	Security unrelated
Vulnerability fix	754	1,226
Non-vulnerability fix	58,979	1,502,994

Note that we are not fighting against the coordinated vulnerability disclosure model since we do not leak information about the vulnerabilities before they are fixed. Different from the full disclosure model, which discloses vulnerabilities too early and puts OSS users at a great disadvantage (since there is no remediation solution available at the time of disclosure), we aim at providing vulnerability fix information to OSS users. In turn, OSS users are able to react to vulnerabilities earlier, to avoid the potential security attacks due to the disclosure latency. We advocate that OSS maintainers should disclose vulnerabilities as soon as the corresponding vulnerability fixes are publicly available. If OSS maintainers need more time to integrate vulnerability fixes into a new release or test the fixes, they should consider postponing submitting the fixes to public code repositories.

C. Other possible application scenarios

Basing on the ability of identifying silent vulnerability fixes, VulFixMiner can be further used to enhance published vulnerability information by locating the corresponding vulnerability fixes. Many vulnerabilities are disclosed without the information of the corresponding fixes and such information is important for security researchers in vulnerability research, for example, automated generation of security patches. In order to locate vulnerability fix information for disclosed vulnerabilities, the existing works mainly rely on the keyword-matching-based approach, which is not able to locate silent vulnerability fixes, while VulFixMiner can. In addition, VulFixMiner is also useful in facilitating some other downstream tasks, for example, the vulnerable dependency alert. Dependabot is a GitHub security feature which is enabled in 3.5m active GitHub repositories [49]. Dependabot analyzes the dependent graph of a project with the published vulnerability information [50], and sends security notifications to OSS users when the vulnerable dependencies are detected. By using VulFixMiner to identify silent vulnerability fixes before the vulnerabilities are disclosed, we believe Dependabot could detect the vulnerable dependencies at an earlier stage.

D. The challenges of leveraging commit messages on mining silent vulnerability fixes

Although the coordinated vulnerability disclosure [1], [2] is commonly and widely applied in OSS,[19,20,21] we still observe many cases of which the information of a vulnerability is leaked before disclosure. For example, the commit message of the fix[22] for CVE-2015-7326 [51] is "patch XXE vulnerability", which explicitly mentions the vulnerability type and the purpose of the commit. Such a practice is not recommended, yet provides an opportunity for identifying vulnerability fixes using commit messages [12]–[14]. To further check how often this "leakage" happens, we apply a regular expression that is used to match security-related words [12] in commit messages. Table VII shows the result of matching. We observe that 38% (754 out of 1,982) of commit messages in vulnerability fixes contain security-related words, indicating the remaining 62% of vulnerabilities were fixed secretly. For such "secret" cases, the message-based approach fails to identify vulnerability fixes. On the other hand, the different commit message documentation styles across projects is also a challenge for identifying vulnerability fixes in a cross-project setting.

[19]https://www.microsoft.com/en-us/msrc/cvd

[20]https://github.com/google/oss-vulnerability-guide

[21]https://www.apache.org/security/committers.html

TABLE VIII: Training and inference time of VulFixMiner.

	VulFixMiner$_J$	VulFixMiner$_P$	VulFixMiner
Training	3 h 47 min	1 h 29 min	4 h 26 min
Inference (Avg.)	0.04 sec/commit	0.02 sec/commit	0.04 sec/commit

E. Time Efficiency

Table VIII shows the time costs of training and average inference for per commit. The training time varies as it depends on the size of the dataset. Once models have been trained, it only takes a few milliseconds to generate the prediction score for a given commit.

VI. Threats To Validity

Internal validity. Threats to internal validity relate to the experimenter bias and errors. The automated approach we proposed for labeling unreported vulnerability fixes in Section IV-B may introduce bias. To mitigate the threat of bias during the labeling, we design and apply a simple yet conservative regular expression, which only matches commits with messages containing "cvd", "nvd", and "vuln". We also randomly sampled 10% of 921 new labeled commits and manually confirmed that all of them are labeled correctly. To mitigate the threat of bias in the manual examination of false-positive cases, we invite security experts who have at least 5 years of experience in software security and express interest in our study. In addition, we give the security experts enough time to conduct these tasks.

External validity. Threats to external validity relate to the generalizability of VulFixMiner. Our dataset is built using disclosed vulnerability fixes, which may not be representative of all vulnerability fixes, especially unreported vulnerability fixes. To mitigate the threat, we designed a strict regular expression to label the unreported vulnerability fixes and include them in our dataset. Also, we have demonstrated the capability of VulFixMiner to mine unreported vulnerability fixes in Section V-A. Future research should include more unreported vulnerability fixes if possible. We also note that we study only

[22]https://github.com/miltonio/milton2/commit/5f81b0c48a817d4337d8b0e99ea0b4744ecd720b

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:46:17 UTC from IEEE Xplore. Restrictions apply.

vulnerability fixes that were reported to CVE. Although the use of CVE is one of the most popular public vulnerability advisory methods, future research should consider including more data sources (e.g., Exploit Database[23] or project-specific web resources), and code or vulnerability related artefacts. However, we evaluate VulFixMiner on a large dataset (i.e., 143,989 commits from 52 projects which are unseen during the training phase) and achieve high performance in a practical setting. Especially, we include two of the most popular programming language, Java and Python in our study. In the future, we plan to include more programming languages.

VII. Related Work

Zhou and Sharma [12] first explored the identification of vulnerability fixes by leveraging commit messages. They extract features using word2vec embedding technique [52] and use ensemble classifiers to classify commits.They first filter out security-unrelated commits using a regular expression including a variety of security-related words. This filtering reduces the imbalanced nature of the data, but also removes secret vulnerability fixes of which messages do not contain security-related words. Chen et al. [14] further considers code changes of commits. By introducing a self-training process, they can also utilize the filtered-out commits. Different from their work, we use deep learning technique in the analysis of code changes in commits. Their tool is not made open source due to its commercial nature and thus we cannot make a comparison with them.

Sabetta and Bezzi [13] used an SVM model, constructed using a BoW representation of code change and commit message tokens, to classify vulnerability fixing commits. Although their approach achieves high performance, the imbalance of their sampled dataset does not reflect the natural imbalance within a realistic scenario. Under the same evaluation setting, VulFixMiner outperforms their approach (see Section IV-F).

Xu et al. [53] proposed SPAIN, a binary-level patch analysis framework, to automatically identify vulnerability fixes. SPAIN locates and identifies the code changes that fix vulnerabilities between two versions (original and patched) of a binary file. Different from SPAIN, VulFixMiner uses commit-level code changes to identify vulnerability fixes, enabling OSS users to discover them in real-time.

Li and Paxson [5] conducted a large-scale empirical study on the development cycle of vulnerability fixes. They observe that vulnerability fixes are poorly timed with public disclosures, and the disclosure time for the majority of fixes is delayed by more than one week. They emphasize the potential risk caused by disclosure delays. Combined with our findings, this further advocates for the importance of real-time, automated identification of vulnerability fixes.

In comparison, we use a deep learning approach to identify vulnerability fixes. We leverage a Transformer-based language model to learn the semantic meaning of code changes.

[23]https://www.exploit-db.com/

VIII. Conclusion and Future Work

In this paper, we propose VulFixMiner, a Transformer architecture-based model for automated secret vulnerability fix mining. To the best of our knowledge, we are the first to use deep learning to identify vulnerability fixes in a practical setting. Specifically, VulFixMiner is the first fine-tuned model on cross-project and cross-language data, using a pre-trained language model, CodeBERT. After fine-tuning, VulFixMiner generates a contextual embedding vector for each commit based on code changes in each affected file. This vector is then used to compute a prediction score, indicating the likelihood that a commit fixes a vulnerability. We evaluate VulFixMiner on the full set of commits from 52 projects, which are never seen during the training phase. The evaluation results show that VulFixMiner outperforms five baselines in discriminative power; VulFixMiner is capable of identifying vulnerability fixes with less inspection effort in practical use. For example, with an inspection effort of reviewing 5% of total LOC, VulFixMiner can identify 49% of the total vulnerability fixes.

Future work could investigate ways to improve our approach, including using additional data, improving contextual embedding vector learning by using a finer granularity of code changes (e.g., at hunk level or line level), as well as improving the generalizability of our approach by involving more programming languages.

References

[1] "ISO/IEC 29147:2018: Security techniques - Vulnerability disclosure," https://www.iso.org/standard/72311.html, 2018.

[2] 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.

[3] "Examining apache struts remote code execution vulnerabilities," https://www.synopsys.com/blogs/software-security/apache-struts-remote-code-execution-vulnerabilities/, 2017, accessed: 2020-04-06.

[4] "Equifax releases details on cybersecurity incident, announces personnel changes," https://investor.equifax.com/news-and-events/press-releases/2017/09-15-2017-224018832, 2017, accessed: 2020-04-06.

[5] 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.

[6] Apache Software Foundation, "CVE-2018-11776," https://nvd.nist.gov/vuln/detail/CVE-2018-11776, 2018.

[7] Semmle Team, "Apache struts vulnerability - CVE-2018-11776," https://blog.semmle.com/remote-code-execution-vulnerability-in-apache-struts-cve-2018-11776/, 2018.

[8] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "OPUS: Online patches and updates for security." in *Proceedings of the 14th USENIX Security Symposium (USENIX Security)*, 2005, pp. 287–302.

[9] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the 4th ACM European conference on Computer systems (EuroSys)*, 2009, pp. 187–198.

[10] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017, pp. 1253–1270.

[11] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda, "Patchdroid: Scalable third-party security patches for android devices," in *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*, 2013, pp. 259–268.

[12] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 11th 2017 joint meeting on foundations of software engineering (FSE)*, 2017, pp. 914–919.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:46:17 UTC from IEEE Xplore. Restrictions apply.

[13] A. Sabetta and M. Bezzi, "A practical approach to the automatic classification of security-relevant commits," in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 579–582.

[14] 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.

[15] 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 11st ACM SIGSAC Conference on Cloud Computing Security Workshop (CCSW)*, 2020, pp. 147–157.

[16] 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.

[17] 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.

[18] "Our replication package," https://github.com/2021-CONFDATA/2021-CONF-DATA, 2020.

[19] Wikipedia, "Full disclosure," https://en.wikipedia.org/wiki/Full_disclosure_(computer_security), 2020, accessed: 2020-04-06.

[20] "Roberta: A robustly optimized bert pretraining approach."

[21] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1073–1085.

[22] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[23] 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 *Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 383–387.

[24] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering (EMSE)*, vol. 21, no. 5, pp. 2072–2106, 2016.

[25] X. Wu, W. Zheng, X. Xia, and D. Lo, "Data quality matters: A case study on data label correctness for security bug report prediction," *IEEE Transactions on Software Engineering (TSE)*, 2021.

[26] 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.

[27] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proceedings of the 37th IEEE/ACM IEEE International Conference on Software Engineering (ICSE)*, vol. 1. IEEE, 2015, pp. 789–800.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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*, 2016, pp. 157–168.

[33] 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.

[34] 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.

[35] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering (TSE)*, vol. 44, no. 5, pp. 412–428, 2017.

[36] G. H. Nguyen, A. Bouzerdoum, and S. L. Phung, "Learning pattern classification tasks with imbalanced data sets," *Pattern recognition*, pp. 193–208, 2009.

[37] A. Severyn and A. Moschitti, "Learning to rank short text pairs with convolutional deep neural networks," in *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval (SIGIR)*, 2015, pp. 373–382.

[38] 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 *Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 34–45.

[39] 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.

[40] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *Journal of Systems and Software*, vol. 83, no. 1, pp. 2–17, 2010.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2014.

[46] R. Caruana, S. Lawrence, and L. Giles, "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping," *Advances in neural information processing systems (NIPS)*, pp. 402–408, 2001.

[47] L. Prechelt, "Automatic early stopping using cross validation: quantifying the criteria," *Neural Networks*, vol. 11, no. 4, pp. 761–767, 1998.

[48] G. K. Rajbahadur, S. Wang, G. Ansaldi, Y. Kamei, and A. E. Hassan, "The impact of feature importance methods on the interpretation of defect classifiers," *IEEE Transactions on Software Engineering (TSE)*, 2021.

[49] GitHub, "Automated security updates," https://github.blog/changelog/2019-11-14-automated-updates/, 2019.

[50] ——, "Keep all your packages up to date with dependabot," https://github.blog/2020-06-01-keep-all-your-packages-up-to-date-with-dependabot, 2020.

[51] MITRE, "CVE-2015-7326," https://nvd.nist.gov/vuln/detail/CVE-2018-11776, 2015.

[52] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS)*, 2013.

[53] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: security patch analysis for binaries towards understanding the pain and pills," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 462–472.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:46:17 UTC from IEEE Xplore. Restrictions apply.