VCMATCH: A Ranking-based Approach for Automatic Security Patches Localization for OSS Vulnerabilities

Shichao Wang[1],Yun Zhang[1‡], Lingfeng Bao[2‡], Xin Xia[2], and Minghui Wu[1]
[1]School of Computer and Computing Science, Zhejiang University City College, Hangzhou, China
[2]College of Computer Science and Technology, Zhejiang University, Hangzhou, China
wangsc98@foxmail.com, {yunzhang, mhwu}@zucc.edu.cn, lingfengbao@zju.edu.cn, xin.xia@acm.org

***Abstract*—Nowadays, vulnerabilities in open source software (OSS) are constantly emerging, posing a great threat to application security. Security patches are crucial in reducing the risk of OSS vulnerabilities. However, many of the vulnerabilities disclosed by CVE/NVD are not accompanied by security patches. Previous research has shown that the auxiliary information in CVE/NVD can aid in the matching of a vulnerability to appropriate commits. The state-of-art research proposed a rank-based approach based on the multiple dimensions of features extracted from the auxiliary information in CVE/NVD. However, this approach ignores the semantic features in the vulnerability descriptions and commit messages, making the model still have room for improvement.**

In this paper, we propose a novel ranking-based approach VCMATCH **(Vulnerability-Commit Match). In addition to extracting the shallow statistical features between the vulnerability and the patch commit,** VCMATCH **extracts the deep semantic features of the vulnerability descriptions and commit messages. Besides,** VCMATCH **applies three classification models (i.e., XGBoost, LightGBM, CNN) and uses a voting-based rank fusion method to combine the results of the three models to generate a better result. We evaluate** VCMATCH **with 1,669 CVEs from 10 OSS projects. The experiment results show that** VCMATCH **can effectively identify security patches for OSS vulnerabilities in terms of *Recall@K* and *Manual Effort@K*, and outperforms the state-of-art model by a statistically significant margin.**

***Index Terms*—Security Patches, Vulnerability Analysis, Mining Software Repository**

I. INTRODUCTION

Open source software (OSS) is widely adopted by many applications in software industry[1]. But OSS vulnerabilities pose a significant risk to software applications. For example, WannaCry ransomware [1] spreads over the Internet by exploiting the National Security Agency's Eternal Blue vulnerability, encrypting computer data on the host computer, and demanding Bitcoin as a ransom. Furthermore, the number of OSS vulnerabilities is continually increasing. According to Georgios' analysis [2], the National Institute of Standards and Technology (NIST) [2] collected more than twice the vulnerability data between 2009 and 2019 than it did between 1999 and 2009. Common Vulnerabilities and Exposures (CVE)[3] has collected over 160,000 vulnerability data so far.

Security patches play an important role in OSS vulnerability management. Firstly, developers can apply patches directly to fix the appropriate vulnerabilities. Second, patches can assist in estimating the impact of a vulnerability (e.g., discovering the software components affected by it) and make a plan to mitigate its risk in a timely manner. Third, developers can analyze the characteristics of a vulnerability based on its patch and learn how to remedy or prevent a similar issue. Additionally, vulnerability patches have been collected to facilitate some research studies. such as vulnerability prediction [3]–[5], vulnerability code clone detection [6], [7], vulnerability testing [8], [9], etc. Therefore, the acquisition of security patches is very important.

However, locating security patches (typically in the form of code commits in a code repository) for a vulnerability is still a challenge [10]. A large number of CVE/NVD entries are also missing security patches. Furthermore, obtaining and identifying patches for vulnerabilities manually is difficult. Hogan et al. [11] reported that manual labeling is a high-skill, time-consuming task and can still be error prone due to lack of knowledge. As a result, it is essential to identify vulnerability patches through an automatic approach.

The auxiliary information in the vulnerability description and the commit message (e.g., CVE ID or bug ID) can be used to automatically match vulnerabilities and patches in code repositories [12]. However, we can only identify patches for a small part of vulnerabilities since such information is usually incomplete and the code repositories contain a large number of code commits. Hence, some researchers propose a machine learning based approach to assist developers in identifying patches for vulnerabilities. Tan et al. [10] proposed a model for matching vulnerabilities and security patches named PatchScout. They transformed the search problem of locating security patches into a ranking problem on code commits. PatchScout considers four categories of features between code commits and vulnerabilities: vulnerability identifier, vulnerability location, vulnerability type, and vulnerability descriptive text. Then, they use the RankNet [13] model to rank code

‡Corresponding author
[1]https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2021.pdf
[2]https://www.nist.gov/
[3]https://cve.mitre.org/

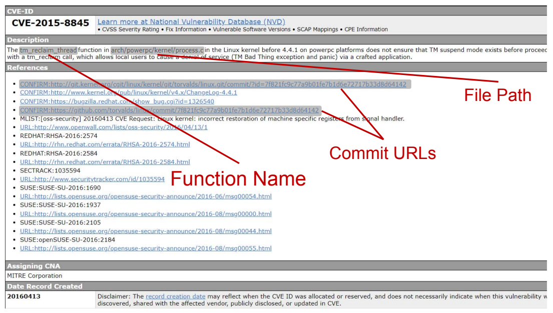
commits. PatchScout can significantly reduce the cost of manual labeling, but it simply extracts the characteristic of the number of words in vulnerability descriptions and commit messages, which lack semantic information contained in the vulnerability description and commit message.

In this study, we propose a novel approach called VCMatch (Vulnerability-Commit Match) to match vulnerabilities and code commits. We follow the study of Tan et al. [10] to rank the code commits based on the correlation between commits and vulnerabilities. The rank position of security patches represents the number of code commits that developers need to manually check. VCMatch combines handcrafted features and deep textual features together to build prediction models. For the handcrafted features, we extract more features to capture correlations between vulnerabilities and code commits in addition to the features used by PatchScout. For example, we compute the time interval between code commit time and CVE-ID assigned time since the time of the code commit for a vulnerability is often close to the CVE-ID assigned time. VCMatch also extracts four dimensions of features including LOC (line of codes) dimension, location dimension, identity dimension, and token dimension. For deep textual features, we utilize BERT [14], one of the state-of-art pre-trained models, to extract semantic correlations between vulnerabilities and code commits. Based on the combination of the handcrafted features and deep textual features, we build three classification models, i.e., XGBoost [15], LightGBM [16], and CNN [17]. These models have shown good performance on imbalanced data as identifying the security patches for a vulnerability from a large number of commits is an imbalanced data task. Finally, VCMatch uses a voting-based ranking fusion method based on the idea of majority voting mechanism. It combines the results of the three classifiers and make a final prediction.

We use 1,669 vulnerabilities in 10 projects to evaluate the performance of VCMatch. We also choose PatchScout and several traditional classifiers as the baselines. The experiment results show that VCMatch can effectively identify security patches for OSS vulnerabilities in terms of *Recall@K* and *Manual Effort@K*. The *Recall@1*, *Recall@10*, and *Manual Effort@10* of VCMatch are 88.86%, 97.06%, and 1.4997, respectively. VCMatch outperforms PatchScout by a statistically significant margin. Our paper makes the following contributions:

1) We build a dataset containing 1,669 vulnerabilities and their corresponding fixing commits from 10 popular OSS projects.
2) We propose a vulnerability-commit matching model VCMatch, which is based on the combination of the handcrafted features and deep textual features[4]. VCMatch also uses a voting-based rank model fusion approach to fuse three classification models to achieve a better performance.
3) We evaluate VCMatch on our built vulnerability-commit matching dataset. The experiment results show that VC-

[4]We release the dataset and the source code of our approach in https://figshare.com/s/0f3ed11f9348e2f3a9f8


CVE-ID

CVE-2015-8845 Learn more at National Vulnerability Database (NVD)
• CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

The tm_reclaim_thread function in arch/powerpc/kernel/process.c in the Linux kernel before 4.4.1 on powerpc platforms does not ensure that TM suspend mode exists before proceec
with a tm_reclaim call, which allows local users to cause a denial of service (TM Bad Thing exception and panic) via a crafted application.

References

- CONFIRM:http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=7f821fc9c77a9b01fe7b1d6e72717b33d8d64142
- CONFIRM:http://www.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.4.1
- CONFIRM:https://bugzilla.redhat.com/show_bug.cgi?id=1326540
- CONFIRM:https://github.com/torvalds/linux/commit/7f821fc9c77a9b01fe7b1d6e72717b33d8d64142
- MLIST:[oss-security] 20160413 CVE Request: Linux kernel: incorrect restoration of machine specific registers from signal handler.
- URL:http://www.openwall.com/lists/oss-security/2016/04/13/1
- REDHAT:RHSA-2016:2574
- URL:http://rhn.redhat.com/errata/RHSA-2016-2574.html
- REDHAT:RHSA-2016:2584
- URL:http://rhn.redhat.com/errata/RHSA-2016-2584.html
- SECTRACK:1035594
- URL:http://www.securitytracker.com/id/1035594
- SUSE:SUSE-SU-2016:1690
- URL:http://lists.opensuse.org/opensuse-security-announce/2016-06/msg00054.html
- SUSE:SUSE-SU-2016:1937
- URL:http://lists.opensuse.org/opensuse-security-announce/2016-08/msg00000.html
- SUSE:SUSE-SU-2016:2105
- URL:http://lists.opensuse.org/opensuse-security-announce/2016-08/msg00044.html
- SUSE:openSUSE-SU-2016:2184
- URL:http://lists.opensuse.org/opensuse-security-announce/2016-08/msg00055.html

Assigning CNA

MITRE Corporation

Date Record Created

20160413 Disclaimer: The record creation date may reflect when the CVE ID was allocated or reserved, and does not necessarily indicate when this vulnerability w
discovered, shared with the affected vendor, publicly disclosed, or updated in CVE.

Fig. 1. An Example CVE Entry

Match outperforms the state-of-art model PatchScout by 17.74%, 7.61%, and 0.979 in terms of *Recall@1*, *Recall@10*, and *Manual Effort@10*, respectively.

Paper organization. Section II introduces the basic concepts related to vulnerabilities and the models used. Section III describes our approach. Section IV presents our experiment setup and results. Section V discusses the implications and threats to validity of our work. Section VI presents related work. Section VII concludes the paper and discusses the future work.

II. Background

In this section, we will introduce vulnerability-related concepts and the background knowledge of pre-trained models used in our model.

A. Vulnerability Concepts

Common Vulnerabilities & Exposures (CVE) is a documented vulnerability information list sponsored by the U.S. Department of Homeland Security. Each CVE entry contains vulnerability-related information, such as CVE-ID, vulnerability description, vulnerability references, and created date (see an example in Fig 1). The CVE-ID is the unique identification of the vulnerability data assigned by the CVE Numbering Authorities (CNAs). The vulnerability description mentions the vulnerable software repository and the consequences caused by the remote attacker. Sometimes the description also mentions function names, file names, and software versions of the vulnerability, which plays an important role in identifying vulnerability patches. Vulnerability references provided by CNA allow the reader to understand better and distinguish vulnerabilities. However, the list is not intended to be complete. Record created date is the CVE-ID assigned to the CNA or the CVE record posted in the CVE list.

National Vulnerability Database (NVD), launched by the National Institute of Standards and Technology (NIST), provides enhanced information based on the CVE list, such as severity scores, fix information, and CWE. CWE (Common Weakness Enumeration) is a category system that identifies vulnerability types with a CWE-ID and a CWE name.

Snyk Vulnerability Database is an open-source vulnerability database launched by Snyk, a company that focuses on open source security. It uses scanning programs to find open source component vulnerabilities in applications. Its vulnerability data has a wide range of sources, including existing NVD and CVE databases, monitoring possible vulnerabilities, pull requests, and other information on Github. Among the 1,669 vulnerabilities used in the study, 112 vulnerabilities' patches are missing in NVD but identified by Snyk. Thus, we collect vulnerability patches from Snyk in this study.

B. BERT

BERT (Bidirectional Encoder Representation from Transformers) [14] has been an emerging natural language processing model in the past few years. Google trained the BERT model on English Wikipedia with the Mask Language Model (MLM) and Next Sentence Prediction (NSP) multitasking and achieved state of the art in 11 downstream natural language processing tasks. The MLM training task enables the BERT model to capture the deep semantics of words, while the NSP training task and the main framework of the algorithm - Transformer, allow the BERT model to more thoroughly capture the bidirectional relationships in utterances and obtain contextual semantics. So we use BERT model in our encoder module to capture the the deep semantics of vulnerabilities and code commits.

C. Prediction Models

We choose to use XGBoost, LightGBM and CNN as basic models, as these models are more effective than traditional machine learning models in software engineering studies [18]–[20], even when the data set is imbalanced [21], [22].

XGBoost is one of the booster algorithms. Its principle is to integrate many weak classifiers to generate a strong classifier according to a particular method. The XGBoost algorithm takes the residuals of the last prediction as the training target for the next tree addition. According to this idea, the model adds weak classifiers sequentially so that the final results are closer to the ground truth values. Each added cart regression tree fits the residuals of the last prediction and keeps splitting and growing according to the features. Eventually, each training sample falls to the corresponding leaf node, and each leaf node corresponds to a value. For a sample, the result of the model prediction is the summation of the corresponding values of that sample overall weak classifiers.

LightGBM also is one of the boosting algorithms. It differs from the XGBoost algorithm in the following aspects. 1) The LightGBM algorithm uses a histogram-based decision tree algorithm, requiring fewer calculations than XGBoost, which needs to calculate the feature gain for each feature traversal. 2) The XGBoost algorithm uses a level-wise decision tree growth strategy, while the LightGBM algorithm uses a leaf-wise growth strategy with depth limitation. The latter can reduce more deviation with the same number of splits but produce deeper decision trees, thus overfitting. 3) The LightGBM algorithm can screen out most of the slight gradient samples according to the gradient of the massive learning data, which can speed up the training speed while maintaining accuracy. 4) The LightGBM algorithm for massive sparse data, according to the conflict degree between data, merges the features with a small conflict degree, changes the sparse matrix into a dense matrix, and reduces the feature dimension.

CNN (Convolutional Neural Network) has a complex network structure. Due to the non-linearity of the activation function, neural networks can detect deeper relationships between training and prediction data. The neural network model generates prediction information through forwarding propagation, obtains the model error through the loss function, and updates the network's internal parameters through backward propagation, improving the model performance.

III. Approach

In this section, we first introduce the overall framework of VCMatch. Then we describe the details of our proposed model.

A. Overall Framework

Figure 2 shows the overall framework of our approach, which consist of five phases: Data Collection, Data Preprocessing, Handcrafted Feature Extraction, Deep Textual Features Extraction, Voting Rank Fusion.

- **Data Collection.** In this phase, we collect code commits from the code repositories, which are hosted in GitHub [23] or GitLab [24]. We collect vulnerability information from CVE/NVD and vulnerability-commit matching data from Snyk.
- **Data Preprocessing.** In this phase, we pre-process the collected data, such as tokenizing textual data, extracting commit ID.
- **Handcrafted Feature Extraction.** In this phase, we extract four dimensions of features that capture the correlations for pairs of vulnerabilities and commits, including LOC, identity, location, and token.
- **Deep Textual Feature Extraction.** In this phase, we use BERT to convert the vulnerability descriptions and commit messages into the encoding features, respectively.
- **Voting Rank Fusion.** In this phase, we build three classification models based on the handcrafted features and deep textual features, and fuse the results of these models to get a final result.

B. Data Collection

We select 10 popular OSS projects, including FFmpeg, ImageMagick, Jenkins, OpenSSL, QEMU, Wireshark, Linux, Moodle, PHP-src, and phpMyAdmin. These projects have been widely studied in many previous studies of vulnerability analysis [25], [26]. To get the commits of these projects, we clone these code repositories from GitHub or GitLab. Table I presents the programming language, the number of CVE entries we collected, and the number of commits in the code repository for each project.

Given a vulnerability, we collect its CVE-ID, textual description, and created date from CVE, and collect its CWE name from NVD. We collect its vulnerability references from the Snyk database. We only retain commit-related URLs by determining whether each URL contains the "commit" keyword or not. For the majority of cases, the reference URLs contain the commit IDs. We use a regular expression to extract commit IDs from links. However, there are a few cases in

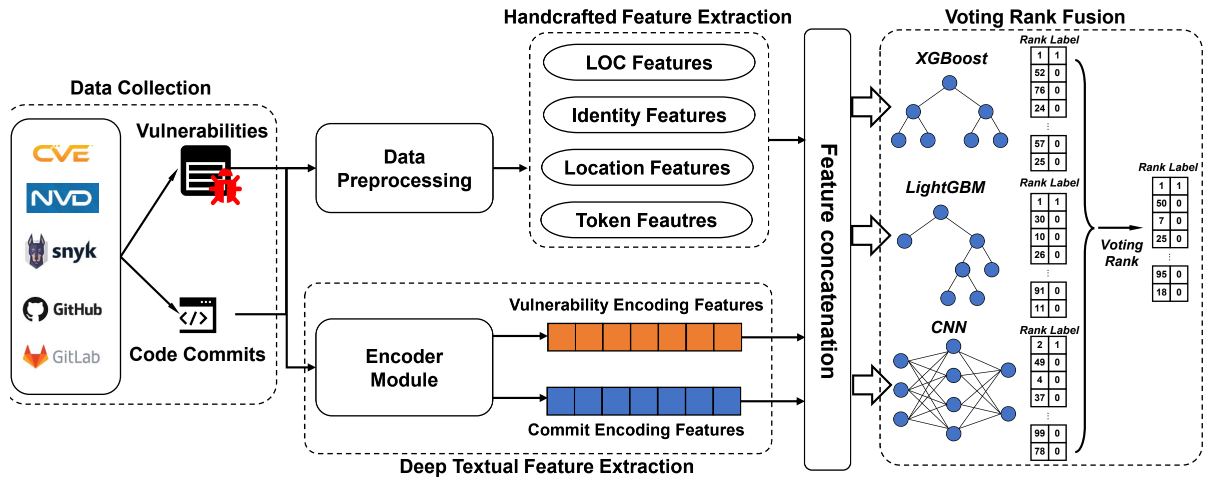


Fig. 2. The Overall Framework of VCMatch

TABLE I
REPOSITORY DETAILS

Software	Language	# CVEs	# Commits
Linux	C/C++	286	1,043,066
php-src	C/C++	153	126,066
phpMyAdmin	PHP	95	122,501
FFmpeg	C/C++	215	103,814
Moodle	PHP	119	102,068
QEMU	C/C++	141	90,309
Wireshark	C/C++	304	82,095
Jenkins	Java	108	31,519
OpenSSL	C/C++	92	30,333
ImageMagick	C/C++	156	19,130
Total		1,669	1,750,901

which the commit IDs are hidden in the URLs. We extract the commit IDs by inspecting the web pages manually.

C. Data Prepropressing

In this study, we only focus on the commits in the master branch of a repository. Developers often create multiple branches to manage workflows of different versions in a project when using Git. For example, there are 31 branches in the FFmpeg repository. The commits in a branch are often applied to multiple branches. For example, developers can use the `git cherry-pick` command to apply a code change to another branch. Most vulnerability patches would be merged into the master branch. Hence, we only search the vulnerability patches in the master branch to save time and computing resources. For the patches in other branches, as they usually have the same code change and commit message, we can identify them easily once we get the vulnerability patch in the master branch.

Given a vulnerability, we first check whether its patches we collected are on the master branch. We observe that a vulnerability might have multiple patches with the same content. So, we only keep the patch on the master branch. If a vulnerability patch is not in the master branch, we try to identify the corresponding patch on the master branch manually. We search on GitHub/GitLab using the commit message to get all the commits with the same commit message, and identify the vulnerability patch on the master branch.

To utilize the textual information, we first pre-process the textual data, including vulnerability descriptions, commit messages, and CWE names, by tokenizing them and removing stop words. Specifically, we use the open-source tokenization method from the Google Cubert repository [27] and the stopword package from the NLTK [28] library to remove stop words. After that, we collect all tokens in the commit messages and vulnerability descriptions and get the token intersection between them. The token intersection is considered "useful tokens". In the later tokenization process, we only keep the useful tokens, which helps to reduce the data size and computation consumption. Other tokens that exist only in vulnerability information or commit information are considered useless and dropped.

D. Handcrafted Feature Extraction

In this section, we introduce the handcrafted features. Table II shows the details of these features. In addition to the features used by PatchScout [10], we extract extra features (shown in bold text in the table). Finally, we extract four dimensions of features to measure the correlation between the input vulnerability and commits, i.e., LOC Dimension, Identity Dimension, Location Dimension, and Token Dimension. Given a pair of a vulnerability and its fixing commit, we extract the following dimensions of features:

LOC Dimension. The vulnerability patches often modify fewer lines of code in comparison to the commits that implement a functional requirement [29]. So, we calculate the number of lines of code (LOC) modified by a code commit, including the LOC added, deleted, and modified in total.

Identity Dimension. Some vulnerability-fixing commits contain the corresponding CVE-ID explicitly, such as the commit in OpenSSL to fix CVE-2016-2107 [30]. Thus, we use a regular expression (i.e., "CVE-[0-9]{4}-[0-9]{1, 8}") to match CVE-ID in the commit messages. Furthermore, a vulnerability may be classified as a common bug and assigned a bug/issue ID. We also extract bug ID and issue ID from the commit messages using a regular expression. Additionally, developers often add an URL of CVEs or bug reports in the commit

TABLE II
HANDCRAFTED FEATURES

Dimension	Feature	Description
LOC Features	**Code Added Num**	# of lines of code added in the commit.
	Code Deleted Num	# of lines of code deleted in the commit.
	Code Modified Num	# of lines of code modified in the commit.
Identity Features	**CVE Num**	# of CVE IDs in commit message.
	Bug Num	# of bug IDs in commit message.
	Issue Num	# of issue IDs in commit message.
	URL Num	# of URLs in commit message.
	CVE Match	Whether the code commit mentions the software-specific CVE-ID in the NVD Page.
	Bug Match	Whether the code commit mentions the Bug-ID in the NVD Page.
	Vulnerability Type Relevance	The relevance of the vulnerability texts between NVD and commit.
	Patch Likelihood	The probability of a commit to be a security patch.
Location Features	**Time Interval**	Time interval between code commit time and CVE-ID assigned time.
	Same Filepath Num	# of filepaths that exist in both code commit and vul description.
	Same Filepath Ratio	# of same filepaths / # of filepaths modified by the code commit.
	Unrelated Filepath Num	# of filepaths that exist in code commit but not mentioned in the vul description.
	Same File Num	# of files that exist in both code commit and vul description.
	Same File Ratio	# of same files / # of files modified by the code commit.
	Unrelated File Num	# of files that appear in code commit but not mentioned in the vul description.
	Same Function Num	# of functions that exist both in the commit diff and vul description.
	Same Function Ratio	# of same functions / # of functions modified by the code commit.
	Unrelated Function Num	# of functions that exist in commit diff but not mentioned in the vul description.
Token Features	**Vul-CWE-Msg Same Num**	# of the same tokens between commit message and CWE name.
	Vul-CWE-Msg Same Ratio	Vul-CWE-Msg Same Num / # of CWE name tokens.
	Vul-Commit Tfidf Similarity	Cosine similarity of vulnerability tfidf and commit tfidf.
	Shared-Vul-Msg-Word Num	# of shared words between vul description and commit message.
	Shared-Vul-Msg-Word Ratio	# of Shared-Vul-Msg-Words / # of words in vul description.
	Max of Shared-Vul-Msg-Word Frequency	The max of the frequencies for all Shared-Vul-Msg-Words.
	Sum of Shared-Vul-Msg-Word Frequency	The sum of the frequencies for all Shared-Vul-Msg-Words.
	Average of Shared-Vul-Msg-Word Frequency	The average of the frequencies for all Shared-Vul-Msg-Words.
	Variance of Shared-Vul-Msg-Word Frequency	The variance of the frequencies for all Shared-Vul-Msg-Words.
	Shared-Vul-Code-Word Num	# of shared words between vul description and code diff.
	Shared-Vul-Code-Word Ratio	# of Shared-Vul-Code-Words / # of words in vul description.
	Max of Shared-Vul-Code-Word Frequency	The max of the frequencies for all Shared-Vul-Code-Words.
	Sum of Shared-Vul-Code-Word Frequency	The sum of the frequencies for all Shared-Vul-Code-Words.
	Average of Shared-Vul-Code-Word Frequency	The average of the frequencies for all Shared-Vul-Code-Words.
	Variance of Shared-Vul-Code-Word Frequency	The variance of the frequencies for all Shared-Vul-Code-Words.

messages. We extract URLs from the commit messages as well.

Location Dimension. We use the following information of the CVE entries to locate the corresponding code changes, i.e., the created time, the file name, file path and function name. The time a vulnerability is disclosed by CVE does not always match with the time a vulnerability-fixing commit is created in the code repository. However, Tan et al. find that most record-created dates are not too far away from the vulnerability patch commit time [10]. Thus, we compute the time interval between the time the code commit was created and the time the CVE-ID was assigned. The vulnerability description may include file names, file paths, and function names, as demonstrated in Figure 1. The code commit essentially modified these files and functions. Therefore, we extract the filenames, file paths, and function names modified in the code commit and count times they appear in the vulnerability description. The more times it appears, the more likely it is that the commit is related to the vulnerability.

Token Dimension. We extract token features from the vulnerability descriptions, CWE names, and commit messages to measure the similarity between the vulnerability and the code commit. Given the useful tokens in the vulnerability description and the commit message, we calculate the number of the same tokens between them and the ratio of the number of the same tokens to the number of the vulnerability description tokens. We also extract these two features between the CWE name and the commit message. In addition, we generate TF-IDF vectors for vulnerability description and commit message, respectively. TF-IDF is widely used in information retrieval and data mining to mine important words. Then, we calculate the cosine similarity between these two vectors, which might indicate the similarity between the vulnerability and the code commit.

E. Deep Textual Feature Extraction

Figure 3 presents the architecture of the encoder module used by VCMATCH. We use BERT [14], one of the state-of-art pre-trained models, to generate encoding features of vulnerability descriptions and commit messages, respectively. BERT can map the textual content into a deep semantic vector space, thus substantially boosting the performance of many natural language processing tasks [31]. Beside, we only need to perform a simple fine-tuning process before applying BERT in downstream tasks. The output of the BERT is a 768-dimensional vector, which is too high for the transitional classifiers used in the study. Thus, we adopt a fully connected

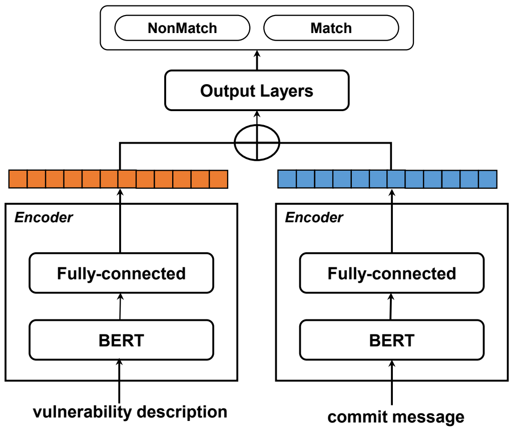


Fig. 3. Encoder Module

layer to compress the vectors generated by BERT into 32 dimensions.

To fine-tune the BERT model, we use vulnerability-commit match and mismatch as the training targets. Vulnerability descriptions and commit messages passed through the encoder module will be encoded into two 32-dimensional encoding vectors. Two vectors then will be spliced together and pass through the output layer, consisting of a fully connected layer and a softmax layer. We use the cross-entropy loss function to narrow the difference between the prediction and the ground truth. Finally, we use the encoder module to generate 32-dimensional encode features for the vulnerability and the code commit, respectively.

F. Voting Rank Fusion

We get 100 features in total for each pair of a vulnerability and its fixing commit after feature extraction, including 36 handcrafted features and 64 deep textual features. Then, we train three classification models (i.e., XGBoost, LightGBM, and CNN) and perform a model fusion to make the final prediction.

Identifying security patches is a classification problem on extremely imbalanced data. Given a vulnerability, only few commits are security patches (a.k.a, positive cases), while the rest of the commits are all negative cases. Traditional machine learning models can easily underfit and fail to separate positive and negative samples correctly. Therefore, we choose XGBoost and LightGBM in this study, which has shown promising performance on imbalanced data [21], [22].

In addition, we choose the CNN model to build a classifier. However, instead of the commonly used loss function for classification models such as cross-entropy, we choose the focal loss function [32], an excellent loss function widely used in recent years to deal with imbalanced image classification tasks. The cross-entropy function is a direct summation of the cross-entropy of each training sample with the same weight. To cope with imbalanced data, a common method is to set the weight values of positive and negative samples, i.e., to set higher weight values for small samples and lower weight values for large samples. However, by setting the weights, we can only control the weights of positive and negative samples, but there is still no way to control the weights of easy and hard to classify samples, so focal loss adopts the use of sample prediction probability as the weight value. The specific formula is as follows, while γ is tunable focusing parameter. Thus, the CNN model can better focus on hard-to-classify data.

$$FocalLoss(p,y) = \begin{cases} -(1-p)^{\gamma}\log(p), & \text{if } y=1 \\ -(p)^{\gamma}\log(1-p), & \text{otherwise} \end{cases} \quad (1)$$

Each classifier predicts a matching probability score for each pair of a vulnerability and its fixing commit in the prediction phase. We use a voting-based ranking fusion method named Voting Ranking to combine the results of the three classifiers. Compared with the data fusion methods commonly used in previous papers [33], VCMATCH based on our voting-based ranking fusion method achieves better performance (see RQ4 in Section IV-D). For a vulnerability, we have the rank of each candidate commit based on the matching probability score for the three classifiers, denoted as $rank_{xgb}$, $rank_{lgb}$, and $rank_{cnn}$. Then we take the two closest values from these three rank values of the commit, denoted as $rank_1$, $rank_2$, representing the two most trustworthy ranks. Finally, we calculate the average of $rank_1$ and $rank_2$, denoted as $rank_{avg}$. We resort the commits based on the $rank_{avg}$ and get the new rank of the commits.

IV. EVALUATION

A. Experiment Setup

We evaluate our proposed approach on the collected dataset that contains 1,669 vulnerabilities from 10 OSS projects (see Section III-B).

To train the prediction models, we use the pairs of these 1,669 vulnerabilities and their corresponding fixing commits as the positive samples. We follow the same way used in the study of PatchScout [10] to construct the negative samples, i.e., we randomly sample 5,000 other commits in the code repository as the negative samples for each positive sample. Specifically, We use the same method to generate features of positive and negative samples. For example, for each sample, we count the number of files that exist in both commit sample and vulnerability description, and use trained encoder module to generate textual information of all samples. In total, our training set has 8,346,669 pairs of vulnerabilities and commits. We use an Ubuntu 18.04 64-bit machine (with 187 GB memory, 4 Intel Xeon Silver 4216 processors and 1 Titan RTX GPU) for model training.

To evaluate the models, we perform a five-fold cross-validation as it is commonly used in many previous papers [34]–[37]. All the vulnerabilities in the dataset are shuffled and equally divided into five folds. The negative samples corresponding to a vulnerability are assigned to the fold that contains it. Training and testing are performed five times (i.e., five runs). For i_{th} run, the i_{th} fold is used as the testing set and the other four folds are combined as the training set. We calculate the mean value of every metric (see Section IV-C) in the five runs as the final result.

TABLE III
RECALLS OF EACH MODEL. STATISTICALLY SIGNIFICANT RESULTS ARE INDICATED WITH *(P-VALUE <0.05) AND **(P-VALUE <0.01).

Top K	1	2	3	4	5	6	7	8	9	10
Logistic Regression	10.43%**	12.70%**	14.56%**	16.06%**	17.44%**	18.87%**	20.13%**	20.97%**	21.99%**	23.13%**
Linear Regression	65.97%**	73.16%**	75.67%**	77.77%**	78.37%**	79.15%**	80.94%**	81.25%**	82.09%**	82.38%**
PatchScout (non-PHP)	**66.31%****	**74.96%****	**78.28%****	**80.55%****	**83.64%****	**84.74%****	**85.63%****	**86.32%****	**86.74%****	**87.49%****
PatchScout	**71.12%****	**77.95%****	**81.43%****	**83.40%****	**86.10%****	**87.06%****	**87.84%****	**88.44%****	**88.80%****	**89.45%****
XGBoost	88.67%*	91.85%*	93.83%*	94.49%*	95.21%*	95.57%*	95.87%*	96.29%*	96.58%*	96.70%*
LightGBM	86.01%**	90.77%**	92.27%**	93.11%**	94.19%**	94.61%**	94.97%**	95.27%**	95.45%**	95.74%**
CNN	85.80%**	89.51%**	90.95%**	91.91%**	92.51%**	93.17%**	93.59%**	94.01%**	94.19%**	94.31%**
VCMATCH (NON-PHP)	**86.80%**	**90.51%**	**92.78%**	**93.61%**	**94.30%**	**94.71%**	**95.12%**	**95.67%**	**96.01%**	**96.29%**
VCMATCH	**88.86%**	**92.03%**	**94.01%**	**94.73%**	**95.33%**	**95.69%**	**96.05%**	**96.52%**	**96.82%**	**97.06%**

TABLE IV
MANUAL EFFORTS OF EACH MODEL

Top K	1	2	3	4	5	6	7	8	9	10
Logistic Regression	**1.0000**	1.8957	2.7687	3.6231	4.4627	5.2882	6.0994	6.8981	7.6884	8.4685
Linear Regression	**1.0000**	1.3403	1.6087	1.8520	2.0743	2.2906	2.4991	2.7028	2.9013	3.0695
PatchScout (non-PHP)	**1.0000**	**1.3314**	**1.5834**	**1.7952**	**1.9841**	**2.1419**	**2.2886**	**2.4263**	**2.5571**	**2.6837**
PatchScout	**1.0000**	**1.2888**	**1.5093**	**1.6950**	**1.8610**	**2.0000**	**2.1294**	**2.2510**	**2.3667**	**2.4787**
XGBoost	**1.0000**	1.1132	1.1947	1.2564	1.3116	1.3595	1.4038	1.4451	1.4823	1.5164
LightGBM	**1.0000**	1.1390	1.2312	1.3085	1.3774	1.4356	1.4895	1.5398	1.5872	1.6327
CNN	**1.0000**	1.1420	1.2468	1.3373	1.4182	1.4931	1.5614	1.6255	1.6854	1.7435
VCMATCH (NON-PHP)	**1.0000**	**1.1260**	**1.2145**	**1.2803**	**1.3377**	**1.3882**	**1.4346**	**1.4768**	**1.5135**	**1.5467**
VCMATCH	**1.0000**	**1.1114**	**1.1911**	**1.2510**	**1.3038**	**1.3505**	**1.3936**	**1.4332**	**1.4679**	**1.4997**

B. Baselines

We implement PatchScout [10] by ourselves as a baseline model since the replication package of PatchScout is not available. PatchScout uses the GumTree tool [38] to get code commit characteristics and generate the Patch Likelihood feature. But GumTree does not support PHP language [39]. So, we set the value of Patch Likelihood to be 0.5 for the commits in the PHP projects. Additionally, we also apply our approach and PatchScout on all the vulnerabilities except those in the PHP projects. And we also performed a 5-fold cross-validation on non-PHP projects to avoid wrong choice of parameter. We also choose the linear regression model, the logistic regression model, XGBoost, LightGBM, and the CNN model as baseline models.

C. Evaluation Metrics

We use top-k recall and manual effort to evaluate the performance of VCMATCH and the baseline models, which is the same as the study of PatchScout.

Recall@K refers to the ratio of the number of patches located in the top-K results to the number of all patches. Higher *Recall@K* score means better performance.

Manual Effort@K refers to the number of results that need to be manually checked to get the right patch. If the right patch is in top-K, the manual effort will be the rank of the patch; otherwise, the manual effort will be K, which represents checking all K results but cannot get the right patch. When K is equal to 1, the values of *Manual Effort* must be equal to 1 since there is only one patch we need to check. *Manual Effort@K* is calculated as follows, while n is the number of test cases: *Manual Effort@K* $= \frac{\sum_{i=1}^{n} min(rank_i, K)}{n}$. Less *Manual Effort@K* score means better performance.

D. Experiment Results

In this section, we present the experiment results by answering the following research questions:

RQ1: Can VCMATCH effectively and efficiently locate security patches for OSS vulnerabilities?

Table III and IV show the results of *Recall@K* and *Manual Effort@K* (K is from 1 to 10) for VCMATCH and the baseline models, respectively. As shown in these two tables, the logistic regression model achieves the worst performance compared with the other models. Its recalls are much smaller than those of the other models, and manual efforts are much larger than those of the other models, indicating that the logistic regression model cannot be used in practice.

In terms of *Recall@K* and *Manual Effort@K*, PatchScout outperforms the linear regression model, but achieves worse performance than the three classifiers used in VCMATCH. And PatchScout (non-PHP) still achieves worse performance than VCMATCH (non-PHP). Those indicates that the features used by VCMATCH can improve the performance of identifying security patches. Among these three classifiers, XGBoost outperforms LightGBM and CNN. Its recalls are more than 90% when K is greater than 1. By leveraging the voting rank method to combine the results of the three classifiers, VC-MATCH achieves the best performance in terms of all metrics. *Recall@1*, *Recall@10*, and *Manual Effort@10* of VCMATCH are 88.86%, 97.06%, and 1.4997, respectively. Compared with PatchScout, VCMATCH has a higher *Recall@1*, *Recall@10*, and *Manual Effort@10*, improving 17.74%, 7.61%, and 0.979, respectively. We apply Wilcoxon signed-rank test [40] with Bonferroni correction [41] to investigate whether VCMATCH has significant improvement compared to PatchScout. We also compute the Cliff's delta. We find that the improvements on *recall@K* and *manual effort@K* are statistically significant($p - values \leq 0.05$) at the confidence level of 95% and of a large effect size on the metrics($|d|$>0.474).

Overall, VCMATCH can effectively and efficiently locate security patches for vulnerabilities and outperform the state-of-art approach PatchScout and the baseline models.

TABLE V
RECALLS OF THE VCMATCH MODELS WITHOUT EACH FEATURE DIMENSION. STATISTICALLY SIGNIFICANT RESULTS ARE INDICATED WITH *(P-VALUE <0.05) AND **(P-VALUE <0.01).

Top K	1	2	3	4	5	6	7	8	9	10
LOC	84.00%**	89.39%**	91.85%**	93.71%*	93.95%**	94.73%*	95.75%*	95.87%*	95.87%*	96.11%*
Identity	83.64%**	89.16%**	90.65%**	92.33%**	93.53%**	93.77%**	94.67%**	94.85%**	94.91%**	95.51%**
Location	65.85%**	75.61%**	78.85%**	80.95%**	82.98%**	83.76%**	85.14%**	85.86%**	85.92%**	86.34%**
Token	79.03%**	83.16%**	86.46%**	88.56%**	89.16%**	90.53%**	91.91%**	92.03%**	92.45%**	92.93%**
Vulnerability_Encode	84.24%**	88.56%**	90.23%**	92.33%**	93.53%**	94.37%**	95.15%**	95.27%**	95.27%**	95.51%**
Commit_Encode	77.65%**	83.76%**	85.44%**	87.36%**	87.78%**	88.74%**	90.11%**	90.23%**	90.47%**	91.31%**
VCMATCH	88.86%	92.03%	94.01%	94.73%	95.33%	95.69%	96.05%	96.52%	96.82%	97.06%

RQ2: How important is each dimension of features used by VCMATCH?

VCMATCH uses two categories of features: handcrafted features and deep textual features. Handcrafted features include four-dimensional features: LOC dimension, identity dimension, location dimension, and token dimension. Deep textual features include vulnerability description encoding features and commit message encoding features. In this research question, we want to investigate the importance of each dimension of features in the vulnerability commit matching process.

Six variants of the VCMATCH model are re-trained, each deleting one feature dimension. We also perform five-fold cross-validation to evaluate these variants of the VCMATCH model. As we find that *Recall@K* and *Manual Effort@K* is consistent in RQ1, we only use *Recall@K* in this RQ. To understand the impact of these dimensions of features, we also apply Wilcoxon signed-rank test [40] with Bonferroni correction [41] to analyze the statistical significance of the improvement of the original VCMATCH on the variant without one feature dimension. We use Cliff's delta to measure the effect size of the improvement.

Table V present the top-k recalls of these variants. The original VCMATCH outperforms all the variant models, indicating that every feature dimension contributes to the VCMATCH model. The location feature dimension is the most discriminative dimension, with the top 1 recall dropping by 23.01% and cannot reach 90% of recall at the top 10. Among the features in location dimension, the information about the file name, file path, and function name in the vulnerability description is useful to locate the corresponding code changes. On the other hand, the time interval can help exclude lots of irrelevant code commits. The commit encoding feature dimension also plays an important role in identifying security patches. The top-1 recall decreases by 11.21% after dropping this dimension of features, indicating the importance of using deep textual features to capture semantic information.

However, the LOC feature dimension and vulnerability encoding feature dimension only slightly improve the performance of VCMATCH. For LOC feature dimension, many common code commits usually have similar lines of code to the security patches, so this dimension can not effectively distinguish whether a code commit is a vulnerability patch or not. For vulnerability encoding feature dimension, the positive and negative samples corresponding to each vulnerability have the same vulnerability encoding features, so those features cannot be effectively used.

Furthermore, we find that the improvement of the original VCMATCH on the variants without one feature dimension is statistically significant($p - values \leq 0.05$) and of a large effect size on the metrics($|d|$>0.474). Thus, the location feature dimension and commit encoding feature dimension are the two most discriminative dimensions. However, using all dimensions of features is better.

RQ3: How effective is VCMATCH in cross-project setting?

In RQ1 and RQ2, we build the prediction models based on all the historical data of the the projects in our dataset. However, a new project may not have enough historical data for building a model. Hence, in this research question, we want to know the effectiveness of VCMATCH on locating security patches for vulnerabilities in a cross-project setting. For each project, we built the VCMATCH model on the data of the other nine projects, then use the model to identify the security patches for this project.

Table VI presents the recalls of cross-project identification for VCMATCH. We sort the projects by *Recall@1* in this table. Except for the project Moodle, Jenkins, and ImageMagick, the top-k recalls of VCMATCH on the other projects are close to or more than 90%, indicating a good performance of VCMATCH in cross-project setting. The recalls of VCMATCH on the Wireshark project even reach 100% when K is greater than 2. For the project Moodle and Jenkins, the recalls of VCMATCH are much smaller than 90% when K is small. But when K becomes larger ($K \geq 7$), the recalls increase to nearly 90%. However, VCMATCH performs poorly for the ImageMagick project, with only 25% recall@1 and 63.46% recall@10. We analyzed the data of the ImageMagick project and found that the underlying reason might be that many commit messages in this project do not contain much useful textual information but only contain an issue or pull request URL. Thus, for many handcrafted features used by VCMATCH such as the location features and the token features, VCMATCH can not extract the important information. Also, the commit encoding features cannot capture the semantic information. Therefore, the VCMATCH model built on the other projects cannot effectively identifying security patches for the vulnerabilities in the ImageMagick project.

Thus, VCMATCH can effectively locate security patches for vulnerabilities in the cross-project setting for most of the projects except ImageMagick.

RQ4: How effective is our ranking fusion method compared with other model fusion methods'?

TABLE VI
RECALLS OF VCMATCH IN CROSS-PROJECT SETTING

Top K	1	2	3	4	5	6	7	8	9	10
Wireshark	98.68%	99.34%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
phpMyAdmin	96.84%	97.89%	97.89%	97.89%	98.95%	98.95%	98.95%	98.95%	98.95%	98.95%
FFmpeg	94.42%	97.21%	98.60%	98.60%	98.60%	98.60%	98.60%	98.60%	98.60%	98.60%
OpenSSL	93.48%	94.57%	94.57%	94.57%	94.57%	96.74%	96.74%	96.74%	96.74%	96.74%
QEMU	89.36%	91.49%	93.62%	94.33%	94.33%	94.33%	95.04%	95.04%	95.04%	95.04%
Linux	87.76%	89.86%	91.96%	93.71%	94.41%	94.76%	95.10%	95.10%	95.10%	95.45%
php-src	87.58%	94.12%	94.77%	95.42%	96.08%	96.08%	96.08%	96.08%	96.73%	96.73%
Moodle	70.59%	80.67%	83.19%	86.55%	86.55%	86.55%	87.39%	87.39%	87.39%	89.08%
Jenkins	58.33%	69.44%	73.15%	78.70%	81.48%	83.33%	87.04%	87.96%	87.96%	89.81%
ImageMagick	25.00%	35.26%	41.67%	44.87%	50.00%	55.13%	57.05%	60.26%	61.54%	63.46%

TABLE VII
RECALLS OF VCMATCH USING DIFFERENT MODEL FUSION METHODS

Top K	1	2	3	4	5	6	7	8	9	10
Borda count	88.02%	92.09%	93.59%	94.79%	95.21%	**95.69%**	95.93%	95.99%	96.11%	96.35%
CombSUM	88.68%	**92.39%**	93.77%	**94.85%**	95.27%	**95.69%**	95.99%	96.41%	96.70%	97.00%
Max	88.50%	92.27%	93.83%	94.73%	95.27%	95.62%	95.99%	96.35%	96.58%	96.76%
Min	87.48%	90.59%	92.09%	92.99%	93.71%	93.95%	94.49%	94.79%	95.03%	95.15%
Voting Rank	**88.86%**	92.03%	**94.01%**	94.73%	**95.33%**	**95.69%**	**96.05%**	**96.52%**	**96.82%**	**97.06%**

VCMATCH uses a new model fusion method named voting rank to combine the results of the three classifiers. However, many model fusion methods have been proposed in previous studies [33]. Thus, in this research question, we want to investigate whether our model fusion method is more effective than the previous fusion methods.

There are six model fusion methods used in the study of Zhang et al. [33], i.e., `Min`, `Max`, `CombANZ`, `CombMNZ`, `CombSUM`, and `Borda count`. The `Min` and `Max` method rank the data based on the smallest and largest probability among base models, respectively. `CombANZ` ranks the data based on the average of non-zero probabilities of the base models. `CombMNZ` ranks the data based on the sum of base models' non-zero probabilities multiply non-zero probabilities' number, while `CombSUM` based on the sum of base models' probability results. `Borda count` first ranks based on each model probability, and then ranks based on the sum of previous rank results. Since there is no non-zero probabilities in our task, the ranking results of `CombANZ`, `CombMNZ`, and `CombSUM` are the same. Thus, we only use `CombSUM` in the evaluation.

We build the VCMATCH models using different model fusion methods. Table VII presents the recalls of the VCMATCH models using these fusion methods. The VCMATCH model based on our voting-based rank fusion method achieves the best performance in terms of all top-k recalls except when K is equal to 2 and 4.

V. DISCUSSION

A. Implications

We have the following implications based on the findings in the study:

More aspects of features can help locating security patches for vulnerabilities. In addition to the features used by PatchScout [10], we use extra features in our proposed model. The experiment results in RQ2 show that these new features can improve the performance of identification of security patches, such as some simple LOC features. So, many other features from different aspects can be used to locate security patches for vulnerabilities, such as the statistical models that capture the naturalness of vulnerable code [42], the information generated by program analysis techniques, and the human factors of developers.

Deep semantic information is useful to match vulnerabilities and code commits. The experiment results in RQ1 and RQ2 show that the deep textual features for vulnerability description and commit messages play an important role in identifying security patches. In the future, we can also use deep learning techniques to represent the code change in commits [43], [44].

More information, more accurate the prediction models of identification of security patches for vulnerabilities is. As shown in RQ3, VCMATCH performs poorly for the ImageMagick project since many commit messages in this project contain less textual description and only have issue or pull request URLs. Thus, to achieve better performance, we need to extract the content from the URLs mentioned in the commit messages or vulnerabilities. Furthermore, we recommend developers provide detailed information in commit messages and vulnerability descriptions, which help locate security patches easily.

B. Threats to Validity.

Internal Validity. One of the threats to validity is the potential errors in the vulnerability-commit matching dataset we collect. There might be cognitive biases when labeling vulnerability patches. Liu et al. [25] found that some vulnerabilities were assigned wrong and inaccurate. To mitigate this threat, we randomly sampled 10% of patch commits and confirmed that all of them are correctly labeled by inspecting the corresponding materials such as documentation and bug reports. Another threat is that there might be errors in the implementation of VCMATCH and the baseline models. We double-checked and fully tested the code, but there could still exist some errors we did not find.

External validity. The main threat to external validity is the generalizability of VCMATCH. We evaluate VCMATCH with 1,669 vulnerabilities from 10 popular OSS projects, covering different programming languages. The number of vulnerabilities involved in our study is also larger than that used in the study of PatchScout (i.e., 685 OSS CVEs). In the future, we plan to evaluate VCMATCH on more vulnerabilities from different kinds of OSS projects.

VI. RELATED WORK

A. Vulnerability Data Collection

In the field of data-driven research, data collection is also one of the focuses of research work. A suitable data set is conducive to the smooth development of research work.With the development of vulnerability research in recent years, many vulnerability data sets to assist research have also emerged.

Mitropoulos et al. [45] statically analyzed the Maven software repositories and established a Java package vulnerability database. Its indicators include the version of the Java package, specific metadata, dependencies, etc. Based on static analysis tools, Zheng et al. [46] used differential analysis to annotate code functions and segments and created a C/C++ data set D2A. Apart from this, most of the data sets are derived from CVE and NVD. Fan et al. [47] crawled data from CVE and constructed a C/C++ data set Big-Vul. Li et al. [48] collected two types of vulnerability (buffer error vulnerability and resource management error vulnerability) from CVE and NVD and created CDG data set. Jimenez et al. [49] proposed VulData7, a vulnerability data collection framework, which can collect four code repository data from NVD. Ponta et al. [50] generated Java data sets from NVD and project-specific web resources. Unlike other data sets that only contain a single programming language, Nikitopoulos et al. [51] created a data set containing more than 40 programming languages. Bhandari et al. [52] proposed a framework which can automatically collect and manage vulnerabilities. The data set contains detailed code and vulnerability information, such as vulnerability type, vulnerability severity level, vulnerability function, and vulnerability file, which reduces the difficulty for developers to obtain data. Xu et al. [53] propose a scalable binary-level patch analysis framework named SPAIN to automatically identify security patches in binaries.After collecting the data, Guan et al. [54] processed the data further by extracting the CFG and DFG of the code and creating the CPG-based vulnerability data set. Researchers can directly apply graph neural networks to predict source code vulnerabilities.

All these works identify the vulnerability information and security patch information, but cannot associate them together. Different from these works, VCMATCH supports locating the security patches of a specified vulnerability.

B. Security Patch Study

The research on vulnerability security patches helps researchers to acknowledge the patches' distribution and development cycle, developers' fixing vulnerability behavior, get vulnerable code to predict vulnerabilities in the software projects and so on.

Bosu et al. [55] find that the majority of vulnerable code changes are written by the senior developers. Nappa et al. [56] discover that the time interval of the same vulnerability patch on different software varies greatly, up to 118 days and the patching rate of different software is also different. Farhang et al. [57] find that the update frequency of severity-level vulnerabilities is more stable, compared with the update frequency of all vulnerability patches. They also find that the patch release data is earlier than the data of disclosure in public data sets for 94% Android vulnerabilities. Xiao et al. [58] study patch signatures and use it to detecting recurring vulnerabilities. Li et al. [59] analyze the whole patch development life cycle. Hwang et al. [60] conduct study on Solidity patches and live contracts, finding many Solidity developers ignore the importance of Solidity patches. Lin et al. [61], Partenza et al. [62], and Zhou et al. [63] get the vulnerability code through vulnerability patches and use abstract syntax trees to predict vulnerabilities. With the development of code cloning detection technology, many researchers [5], [64], [65] use similar vulnerability codes to predict vulnerabilities.

These work study and learn about security patches, or obtain the vulnerability code based on patches to predict the vulnerability. We conduct the patch study from a new perspective. We train a vulnerability patch identification model based on existing vulnerability patches to help developers discover patches.

VII. CONCLUSION AND FUTURE WORK

The matching of vulnerabilities and security patches is conducive to researchers to carry out better vulnerability-related research, and it also helps developers understand how to fix various vulnerabilities correctly. In this paper, we propose a model named VCMATCH, which based on six-dimensional features. According to the experiment results, the combination of shallow statistical features and deep semantic features dramatically improves the effectiveness of the VCMATCH model. We can find that our VCMATCH model has achieved great success through comparative experiments and is better than the existing vulnerability-commit matching methods. Besides, the VCMATCH model also achieves good results in cross-project forecasting. In future research work, we plan to expand the data through web content such as bug reports and GitHub issues. Vulnerability references often involve bug reports, GitHub issue links. These links may contain data related to vulnerabilities or software code. Crawling and reasonably extracting critical data can further enrich the commit data, enhancing the model's effect on software repositories lacking commit messages.

ACKNOWLEDGEMENT

This research was supported by the Natural Science Foundation of Zhejiang Province under Grant No. LQ21F020008, and the National Science Foundation of China (No. 62141222, No. U20A20173 and No. 6190234).

References

[1] S. Algarni, "Cybersecurity Attacks: Analysis of "WannaCry" Attack and Proposing Methods for Reducing or Preventing Such Attacks in Future," in *ICT Systems and Sustainability*, M. Tuba, S. Akashe, and A. Joshi, Eds. Singapore: Springer Singapore, 2021, pp. 763–770.

[2] G. Aivatoglou, M. Anastasiadis, G. Spanos, A. Voulgaridis, K. Votis, D. Tzovaras, and L. Angelis, "A rakel-based methodology to estimate software vulnerability characteristics & score-an application to eu project echo," *Multimedia Tools and Applications*, pp. 1–21, 2021.

[3] S. Rasheed and J. Dietrich, "A hybrid analysis to detect Java serialisation vulnerabilities," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1209–1213.

[4] Xiao Yang, Xu Zhengzi, Zhang Weiwei, Yu Chendong, Liu Longquan, Zou Wei, Yuan Zimu, Liu Yang, Piao Aihua, and Huo Wei, "VIVA: Binary Level Vulnerability Identification via Partial Signature," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Honolulu, HI, USA: IEEE, Mar. 2021, pp. 213–224. [Online]. Available: https://ieeexplore.ieee.org/document/9425910/

[5] F. P. Viertel, W. Brunotte, D. Strüber, and K. Schneider, "Detecting Security Vulnerabilities using Clone Detection and Community Knowledge." in *SEKE*, 2019, pp. 245–324.

[6] Z. Liu, Q. Wei, and Y. Cao, "Vfdetect: A vulnerable code clone detection system based on vulnerability fingerprint," in *2017 IEEE 3rd Information Technology and Mechatronics Engineering Conference (ITOEC)*. IEEE, 2017, pp. 548–553.

[7] B. Bowman and H. H. Huang, "Vgraph: A robust vulnerable code clone detection system using code property triplets," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 53–69.

[8] H. Yang, D. Liang, X. Kuang, and C. Xu, "A vulnerability test method for speech recognition systems based on frequency signal processing," in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*. IEEE, 2018, pp. 943–947.

[9] F. Lebeau, B. Legeard, F. Peureux, and A. Vernotte, "Model-based vulnerability testing for web applications," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2013, pp. 445–452.

[10] X. Tan, Y. Zhang, C. Mi, J. Cao, K. Sun, Y. Lin, and M. Yang, "Locating the Security Patches for Disclosed OSS Vulnerabilities with Vulnerability-Commit Correlation Ranking," in *Proceedings of the 28th ACM Conference on Computer and Communications Security(CCS)*. ACM, 2021.

[11] K. Hogan, N. Warford, R. Morrison, D. Miller, S. Malone, and J. Purtilo, "The Challenges of Labeling Vulnerability-Contributing Commits," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2019, pp. 270–275.

[12] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 595–614.

[13] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to rank using gradient descent," in *Proceedings of the 22nd international conference on Machine learning*, 2005, pp. 89–96.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[15] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[16] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, pp. 3146–3154, 2017.

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

[18] S. Wang, P. Dong, and Y. Tian, "A novel method of statistical line loss estimation for distribution feeders based on feeder cluster and modified xgboost," *Energies*, vol. 10, no. 12, p. 2067, 2017.

[19] X. Cheng, N. Liu, L. Guo, Z. Xu, and T. Zhang, "Blocking bug prediction based on xgboost with enhanced features," in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2020, pp. 902–911.

[20] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 318–328.

[21] F. C. Luiz, B. R. de Oliveira Rodrigues, and F. S. Parreiras, "Machine learning techniques for code smells detection: an empirical experiment on a highly imbalanced setup," in *Proceedings of the XV Brazilian Symposium on Information Systems*, 2019, pp. 1–8.

[22] R. Malhotra and K. Lata, "An empirical study on predictability of software maintainability using imbalanced data," *Software Quality Journal*, vol. 28, no. 4, pp. 1581–1614, 2020.

[23] "Build software better, together." [Online]. Available: https://github.com

[24] "Iterate faster, innovate together." [Online]. Available: https://about.gitlab.com/

[25] B. Liu, G. Meng, W. Zou, Q. Gong, F. Li, M. Lin, D. Sun, W. Huo, and C. Zhang, "A large-scale empirical study on vulnerability distribution within projects and the lessons learned," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 1547–1559. [Online]. Available: https://dl.acm.org/doi/10.1145/3377811.3380923

[26] M. Jimenez, M. Papadakis, and Y. Le Traon, "An empirical analysis of vulnerabilities in openssl and the linux kernel," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2016, pp. 105–112.

[27] "google-research/cubert at master · google-research/google-research." [Online]. Available: https://github.com/google-research/google-research

[28] "NLTK :: Natural Language Toolkit." [Online]. Available: https://www.nltk.org/

[29] X. Wang, S. Wang, K. Sun, A. Batcheller, and S. Jajodia, "A machine learning approach to classify security patches into vulnerability types," in *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2020, pp. 1–9.

[30] "The fixing commit for cve-2016-2107," https://github.com/openssl/openssl/commit/68595c0c2886e7942a14f98c17a55a88afb6c292, accessed: 2021-10-19.

[31] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang, "Pre-trained models for natural language processing: A survey," *Science China Technological Sciences*, pp. 1–26, 2020.

[32] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.

[33] Y. Zhang, D. Lo, X. Xia, G. Scanniello, T.-D. B. Le, and J. Sun, "Fusing multi-abstraction vector space models for concern localization," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2279–2322, 2018.

[34] W. Melicher, C. Fung, L. Bauer, and L. Jia, "Towards a lightweight, hybrid approach for detecting dom xss vulnerabilities with machine learning," in *Proceedings of the Web Conference 2021*, 2021, pp. 2684–2695.

[35] J. D. Pereira, J. R. Campos, and M. Vieira, "An exploratory study on machine learning to combine security vulnerability alerts from static analysis tools," in *2019 9th Latin-American Symposium on Dependable Computing (LADC)*. IEEE, 2019, pp. 1–10.

[36] M. Grigoriou, K. Kontogiannis, A. Giammaria, and C. Brealey, "Report on evaluation experiments using different machine learning techniques for defect prediction," in *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*, 2020, pp. 123–132.

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

[38] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[39] "The programming languages supported by gumtree," https://github.com/GumTreeDiff/gumtree/wiki/Languages, accessed: 2021-10-19.

[40] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[41] H. Abdi *et al.*, "Bonferroni and šidák corrections for multiple comparisons," *Encyclopedia of measurement and statistics*, vol. 3, pp. 103–107, 2007.

[42] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the" naturalness" of buggy code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 428–439.

[43] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[44] W. Wang, G. Li, S. Shen, X. Xia, and Z. Jin, "Modular tree network for source code representation learning," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–23, 2020.

[45] D. Mitropoulos, G. Gousios, P. Papadopoulos, V. Karakoidas, P. Louridas, and D. Spinellis, "The Vulnerability Dataset of a Large Software Ecosystem," in *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. IEEE, 2014, pp. 69–74.

[46] Y. Zheng, S. Pujar, B. Lewis, L. Buratti, E. Epstein, B. Yang, J. Laredo, A. Morari, and Z. Su, "D2a: A dataset built for ai-based vulnerability detection methods using differential analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 111–120.

[47] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "AC/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.

[48] Li Zhen, Zou Deqing, Xu Shouhuai, Ou Xinyu, Jin Hai, Wang Sujuan, Deng Zhijun, and Zhong Yuyi, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[49] M. Jimenez, Y. Le Traon, and M. Papadakis, "[Engineering Paper] Enabling the Continuous Analysis of Security Vulnerabilities with VulData7," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 56–61.

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

[51] G. Nikitopoulos, K. Dritsa, P. Louridas, and D. Mitropoulos, "CrossVul: a cross-language vulnerability dataset with commit data," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1565–1569.

[52] G. P. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software," *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 30–39, Aug. 2021, 0 citations (Semantic Scholar/arXiv) [2021-09-22] 0 citations (Semantic Scholar/DOI) [2021-09-22] arXiv: 2107.08760. [Online]. Available: http://arxiv.org/abs/2107.08760

[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 *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017.

[54] Z. Guan, X. Wang, W. Xin, and J. Wang, "Code Property Graph-Based Vulnerability Dataset Generation for Source Code Detection," in *International Conference on Frontiers in Cyber Security*. Springer, 2020, pp. 584–591.

[55] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: An empirical study," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 257–268.

[56] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The attack of the clones: A study of the impact of shared code on vulnerability patching," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 692–708.

[57] S. Farhang, M. B. Kirdan, A. Laszka, and J. Grossklags, "Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities," *arXiv preprint arXiv:1905.09352*, 2019.

[58] 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 *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1165–1182.

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

[60] S. Hwang and S. Ryu, "Gap between theory and practice: An empirical study of security patches in solidity," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 542–553.

[61] Lin Guanjun, Zhang Jun, Luo Wei, Pan Lei, and Xiang Yang, "POSTER: Vulnerability discovery with function representation learning from unlabeled projects," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2539–2541.

[62] Partenza Garrett, Amburgey Trevor, Deng Lin, Dehlinger Josh, and Chakraborty Suranjan, "Automatic Identification of Vulnerable Code: Investigations with an AST-Based Neural Network," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2021, pp. 1475–1482.

[63] Zhou Yaqin, Liu Shangqing, Siow Jingkai, Du Xiaoning, and Liu Yang, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *arXiv preprint arXiv:1909.03496*, 2019.

[64] Q. Hum, W. J. Tan, S. Y. Tey, L. Lenus, I. Homoliak, Y. Lin, and J. Sun, "CoinWatch: A clone-based approach for detecting vulnerabilities in cryptocurrencies," in *2020 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2020, pp. 17–25.

[65] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 896–899.