2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications (SERA) | 979-8-3503-4588-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SERA57763.2023.10197730

Commit Message Can Help: Security Patch Detection in Open Source Software via Transformer

Fei Zuo[†], Xin Zhang[‡], Yuqi Song[‡], Junghwan Rhee[†], Jicheng Fu[†]
[†]University of Central Oklahoma, Edmond, Oklahoma, 73034, USA
{fzuo, jrhee2, jfu}@uco.edu
[‡]University of South Carolina, Columbia, South Carolina, 29208, USA
{xz8, yuqis}@email.sc.edu

***Abstract*—As open source software is widely used, the vulnerabilities contained therein are also rapidly propagated to a large number of innocent applications. Even worse, many vulnerabilities in open-source projects are secretly fixed, which leads to affected software being unaware and thus exposed to risks. For the purpose of protecting deployed software, designing an effective patch classification system becomes more of a need than an option. To this end, some researchers take advantage of the recent advancements in natural language processing to learn both commit messages and code changes. However, they often incur high false positive rates. Not only that, existing works cannot yet answer how much the textual description (such as commit messages) alone can influence the final triage. In this paper, we propose a Transformer based patch classifier, which does not use any code changes as inputs. Surprisingly, the extensive experiment shows the proposed approach can significantly outperform other state-of-the-art work with a high precision of 93.0% and low false positive rate. Therefore, our research further confirms the critical importance of well-crafted commit messages for the later software maintenance. Finally, our case study also identifies 48 silent security patches, which can benefit those affected software.**

***Index Terms*—security patch, open source software, vulnerability detection, transformer**

I. Introduction

In the recent decade, we have witnessed a sharp increase in the number of serious vulnerabilities discovered in Open Source Software (OSS). The Common Vulnerabilities and Exposures (CVE) system maintained by MITRE tracks the publicly known software vulnerabilities. In 2022, 25,059 CVE records were published which is over 4.7 times higher than the number 5,288 reported in 2012 [1].

When a new vulnerability is discovered, a project maintainer or developer needs to timely fix the problem by submitting a patch commit to the code repository. Herein, the patch commit is a specific type of commit, which includes changes to the codebase in response to a particular issue. Furthermore, if the patch commit is to address a vulnerability, we consider it as a security patch. The major motivation for the security patch detection is to mine the those silently fixed vulnerabilities.

The prior studies [2], [3] show that there exist tons of vulnerabilities in OSS projects that are not publicly disclosed with CVEs. Another undeniable fact is that ubiquitous code sharing practices in the software development significantly expedite the spread of vulnerabilities. As a result, the unignorable risk caused by those secretly fixed bugs is, even if these bugs are fixed in a timely manner by the project maintainers, other software which re-use the vulnerable code snippets are left in the dark. Hence, identifying these silent security patches plays a key role in the modern software eco-system. Unfortunately, though the detection of security patches is crucial, this problem is still far from being solved. One straightforward solution is possibly to use manual verification, but it is unrealistic because of the massive time consumption and high labor cost. Due to the gap between the increasing number of vulnerabilities and the limited manpower, automated security patch detection becomes an advisable option.

Traditional software analysis techniques such as symbolic execution [4], [5] and fuzzing [6] have been widely used in the vulnerabilities discovery. However, these techniques usually cannot be applied to incomplete code fragment like the code changes in a commit. Thus, they are not suitable for security patch detection. Not only that, these code-oriented methods also cannot support the commit message analysis, which is in the form of natural language.

Inspired by the tremendous success of applied machine learning and deep learning in diverse areas [7]–[10], many data-driven approaches are proposed for automated security patch detection. For example, some previous works [11], [12] rely on statistic features to describe code changes such as the number of modified lines and the number of modified characters. However, the feature engineering heavily requires heuristics or domain knowledge. Furthermore, as deep learning makes remarkable achievements in natural language processing (NLP), many researchers start to shift their focus towards this new arsenal of tools. As a result, we have observed many works [13]–[15] adopt NLP techniques to analyze both the commit message and the code changes. By this means, the obtained precision can reach around 87.9%.

However, we argue that textual descriptions such as commit messages and comments have contained fruitful and important information. Hence, it is worth exploring to what degree they can help distinguish security patches from others. To this end, we propose a Transformer based patch detector, which only digests the textual description text, involving commit messages, as well as comments associated with the code if there is any. More than that, we are aware of commit messages often include object identifiers (e.g., a specific function name)

979-8-3503-4588-9/23/$31.00 ©2023 IEEE
SERA 2023, May 23-25, 2023, Orlando, USA

and reference links (e.g., a git issue number or a URL), which cannot be simply considered as the natural language text. Therefore, we also design the pre-processing scheme based on regular expressions to handle these noise. Our extensive experiments on a dataset collected from GitHub demonstrate not only the effectiveness of the proposed approach but also the critical importance of well-crafted commit messages for the later software maintenance. Our contributions are summarized as follows:

- To the best of our knowledge, this is the first work on security patch detection with targeting textual descriptions alone. The research outcome fully indicates well-crafted commit messages are very helpful in distinguishing security patches.
- A lightweight text processing method is proposed to handle the irregularity and particularity of the textual contents from patch commits.
- Our extensive experiments demonstrate the effectiveness of the proposed approach, which can reach an accuracy of 93.0%, which outperforms other state-of-the-art works.
- We also identify 48 silent security patches by adopting our prototype system.

The rest of this paper is organized as follows. First, we introduce necessary background knowledge in Section II. Section III presents our approach to patch commit classification, the quantitative evaluation of which is presented in Section IV. Next, the related work is briefly reviewed in Section V. In Section VI, we discuss some limitations and outline the research directions we intend to pursue in the future. Finally, Section VII draws the conclusions.

II. Background

In this section, we briefly introduce the background of patch commit, word embedding, and transformer that we will use in this paper.

A. Commit

Nowadays, git commits are playing a more and more indispensable role throughout the various phases of software development life cycle. In the software maintenance, an update often includes numerous patch commits. Some patches are used to fix vulnerabilities, thus they are called security patches. By contrast, others, i.e. non-security patches, may serve for different purposes such as the performance improvement.

In general, a commit usually consists of a commit message and a diff, which records code changes between different software versions. Fig. 1 is a concrete commit example for the security patch to fix vulnerability CVE-2020-14354. We present this sample commit to visually illustrate each component in a commit. The commit message and the patch diff is separated by a line with a three-dash line. In the diff, we can see the source code file "*ares_getaddrinfo.c*" is modified. Moreover, the line beginning with "`@@`" in a chunk is in red font, which is to display some specific information, i.e. the start changed line number and the function name in the original file. The followings are code changes between the original file and the updated file, where the line marked with "−" means such a line is removed from the previous version, while the line marked with "+" means that is a new line added in the current version.

```
From 1cc7e83c3bdfaafbc5919c95025592d8de3a170e
From: Brad House <brad@brad-house.com>
Date: Thu, 7 May 2020 07:02:31 -0400
Subject: [PATCH] Prevent possible double-free in
ares_getaddrinfo() if ares_destroy() is called

In the event that ares_destroy() is called prior
    to ares_getaddrinfo() completing,
it would result in an invalid read and double-
    free due to calling end_hquery() twice.

Reported By: Jann Horn @ Google Project Zero
---
 ares_getaddrinfo.c | 1 +
 1 file changed, 1 insertion(+)

diff --git a/ares_getaddrinfo.c
b/ares_getaddrinfo.c
index 316fa5a61..be168068b 100644
--- a/ares_getaddrinfo.c
+++ b/ares_getaddrinfo.c
@@ -548,6 +548,7 @@ static void host_callback
    (void *arg, int status, int timeouts,
   else if (status == ARES_EDESTRUCTION)
     {
       end_hquery(hquery, status);
+      return;
     }

   if (!hquery->remaining)
```

Fig. 1. A security patch commit sample for CVE-2020-14354.

Though a diff may show you the code is changed, only the commit message can properly tell the reason for such a change. Consequently, the open source community has recognized the importance of a well-crafted commit message as an effective communication way concerning code changes among the contributors and maintainers of OSS projects. A commit message is comprised of a subject, body, and footer, with both the body and footer being optional. For example, in Fig. 1, the commit message is in purple color.

In particular, the subject is a single line that best summarizes the changes made in the commit. By default, the subject of a single patch starts with "`Subject: [PATCH]`". But if there exist multiple patches, the subject prefix will instead be "`Subject: [PATCH n/m]`". Sometimes when the change is so simple that no further explanation is necessary, a single subject line is fine. Thus, not every commit includes a body. However, if a commit deserves a more thorough description, the body text can provide more details regarding the changes made in the commit. The body is separated by the subject using a blank line. As the last component of a commit message, the footer text is found immediately below the body and is a preferable place to reference issues related to the commit changes. Between the footer and the body, there also will be a blank line as the separator.

B. Word Embedding

In natural langauge processing, word embeddings are frequently used, which learn words numeric representation in a high-dimensional space, to facilitate the further processing in machine learning models. In particular, a word embedding is to capture the semantic meaning of the given word according to its contexts; therefore, the words that share similar contextual semantics are inclined to have approximate numeric representations in the embedding space.

Among the existing techniques, Mikolov's skip-gram model is popular because of its efficiency and low memory consumption [16]. This unsupervised approach learns word embeddings by using a neural network. In detail, given a centre word w_t in the context C_t, the probability function of another word w_k appears in the same context is as (1)

$$P(w_k \in C_t | w_t) = \frac{\exp\left(\mathbf{w}_t' \mathbf{w}_k\right)}{\sum_{w_i \in C_t} \exp\left(\mathbf{w}_t' \mathbf{w}_i\right)} \tag{1}$$

where $\mathbf{w}_k$, $\mathbf{w}_t$, and $\mathbf{w}_i$ are the embeddings of words w_k, w_t, and w_i, respectively.

Thus, given an arbitrary word w_k, its numeric representation w_k is used as a feature vector in the softmax function parameterized by w_t. When trained on a sequence of T words, the model optimization is to maximize the average log probability $J(w)$ as showed in (2)

$$J(w) = \frac{1}{T} \sum_{t=1}^{T} \sum_{w_k \in C_t} \log P(w_k | w_t) \tag{2}$$

where the larger context C_t leads to a higher accuracy. However, when the size of C_t becomes larger, it would be very expensive to optimize $J(w)$, considering the denominator $\sum_{w_i \in C_t} \exp\left(\mathbf{w}_t' \mathbf{w}_i\right)$ goes through all words w_k in C_t. To reduce the training cost but also achieve high-quality word embeddings, in this paper we adopt the skip-gram with negative sampling model (SGNS), which is a popular solution proposed in [17].

C. Transformer

The Transformer [18] is a deep learning model architecture introduced by Google researchers for natural language processing (NLP) tasks. This model architecture revolutionizes the use of self-attention to compute representations of its input and output without using recurrence and convolutions. The Transformer has demonstrated superior performance over existing models on a variety of NLP tasks.

In general, the Transformer is essentially a sequence-to-sequence model, which consists of an encoder and a decoder component. The encoder takes an input sequence and produces a sequence of hidden representations, while the decoder takes the hidden representations and generates an output sequence. Each component is composed of a stack of identical layers, with each layer consisting of a multi-head self-attention mechanism and a position-wise feed-forward neural network. They allow the model to weigh the importance of different positions in the input sequence, so the model's capability of capturing long-range dependencies is remarkably enhanced. In addition, the Transformer architecture incorporates residual connections [8] and layer normalization, which help stabilize training and improve performance. Overall, the Transformer has been a significant advancement in the field of NLP, demonstrating the power of attention mechanisms in deep learning models.

III. Approach

In this section, we introduce an overview of our approach and detail each component in the proposed system.

A. Overview

Our approach consists of two phases, i.e. the training phase and the detection phase. In the former phase, both a Word2vec model and a Transformer based classifier need to be trained using a set of patch commits collected from GitHub[1], which is the largest platform to offer hosting service for software development and version control using Git. As shown in Fig. 2, we first extract textual information from the patch commits, on which the pre-processing is applied then. After this step, a Word2vec model is trained through an unsupervised learning procedure. Based on this, the sequence of words in corpus can be embedded as a numeric matrix, which will be fed into the neural network. Hence, after optimized by a gradient descent based solver such as Adam [19], a well-trained classifier can be eventually obtained. In the detection phase, by pre-processing the unseen patch commit, and inputting the result into the classifier, whether the patch is security related can be determined.

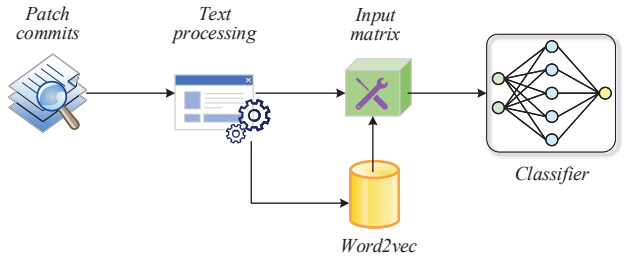

Fig. 2. An overview of the proposed system.

B. Text Processing

First, we need to extract textual contents from a patch commit. In detail, the textual contents consist of two components, that is the commit message and comments in the source code. Because the subject always stars with "`Subject: [PATCH]`" or "`Subject: [PATCH n/m]`", we can use a regular expression pattern,

```
^Subject:␣\[PATCH\]|Subject:␣\[PATCH␣\d+/\d+\]
```

to locate the starting line. The delimiter between the commit message and the patch diff, a three-dash line, can be used as the indicator of the message termination. Similarly, the

[1]https://github.com/

comments contained in the source code are extracted via the regular expression based method as well. All words are lemmatized using WordNet Lemmatizer provided in Natural Language Toolkit (NLTK) [20].

More importantly, the textual descriptions of a patch commit still contain some obscure and special string literals, though they are in the form of natural languages. Those literals not only increase the vocabulary size of the Word2vec model, but also bring noise to data. For example, a commit message can contain URLs to the external references, which are less meaningful for the later semantics analysis. In addition, SHA values, CVE numbers, and some identifiers such as specific function names, which are in the format of a long sequence of characters, are often included in commit messages according to our experience. However, linguistic research indicates that 99.87% of English words contain no more than 15 letters [21]. To avoid the degradation of model quality caused by these obscure literals, we specifically design a set of systematic rules to clean the data, as shown in Table I.

TABLE I
RESPONSE APPROACHES FOR DIFFERENT LITERALS

Type of literal	Response approach
URL	Removed
Email address	Removed
CVE number	Replaced with `CVEXXXXYYYY`
Numeric literal	Replaced with `THISISANUM`
Long string ($size > 15$)	Replaced with `XXXLONGXXX`

C. Input Generation

We adopt the facilities provided by Gensim [22], a popular open-source Python library for the natural language embedding, to implement our Word2vec model. As a result, there are 4,621 words in the vocabulary, each word embedding is a numeric vector with 50 dimensions. Our empirical study shows 94.8% textual descriptions have less than 200 words, so we consider at most first 200 words in each textual content. In particular, for those text segments which have less than 200 words, padding with zeros will be done. As a result, for each textual content, we can generate a real matrix $\mathcal{M} \in \mathbb{R}^{200\times 50}$ as the numeric representation.

D. Classifier

The intact Transformer model consists of an encoder and a decoder, where the encoder component is used to extract semantics from the input and generate a fixed-size representation. By contrast, the decoder component is responsible for generating the output sequence. For example, in a machine translation task, the decoder will generate the resulting sentence in a target language. However, our focus is to build a classifier, thus we only adopt the encoder component of a Transformer model to obtain the semantic embedding for the input. Then, the semantic embedding is passed to a feedforward neural network to make the final prediction. The network architecture is shown as Fig. 3. In particular, the positions embedding is a fixed vector that is added to the input embedding for each token in the input sequence. The vector will be learned during training, which is for the purpose of keeping the information about the position of the token in the sequence. By introducing the positions embedding, the Transformer model is able to take into account the order of the input sequence.

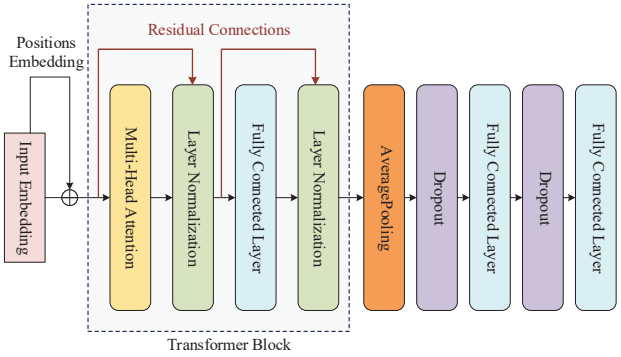

Fig. 3. Network architecture.

IV. EVALUATION

In this section, we first elaborate on the evaluation setup including the dataset, the evaluation metrics, and the environment configurations. Then, we present the evaluation results. The silent security patches discovered in this study are discussed at last.

A. Evaluation Setup

Dataset. Each entry in the CVE database includes a CVE number, a succinct description of the vulnerability, and references (usually in the form of website addresses to external resources). In particular, for the project hosted on GitHub, the website address referring to a security related commit has a format as

```
https://github.com/owner/repo/commit/commit_hash
```

where the `commit_hash` is a SHA1 value that is used as the unique identifier of a commit. Therefore, we can crawl all the reference website addresses in CVE entries and filter out those matching the above rule. After this step, by appending a suffix "`.patch`" at the end of one website address, we can get a patch URL. The corresponding patch file can be retrieved by `wget` according to the given patch URL. For example,

```
https://github.com/jasper-software/jasper/commit/
    fd564ee3377d9fc2484c657e4f464a3fb9764d31.patch
```

is the patch URL for CVE-2021-27845. Furthermore, non-security patches, which are not connected with any CVE records, are randomly retrieved from GitHub by our crawler. We also manually double-check the those patches to avoid unreported security patches.

It is worth noting that, we concentrate on patches of C/C++ projects because they are very common in the open source

ecosystem. Moreover, our research question is to explore how much the textual descriptions can influence the security patch classification, therefore the patch commits that don't include any commit messages will be overlooked. Finally, we collect 6,226 patches that belong to projects hosted on GitHub, in which 2,488 samples are positive (i.e., security patches) and 3,738 samples are negative. We randomly choose 90% (i.e. 5,603) of our dataset as the training set and the remaining 10% (i.e. 623 samples) as the testing set.

Evaluation Metrics. $Precision$, $Recall$, $F_1\ score$, and $Accuracy$ are widely used evaluation metrics for classification systems. $Precision$ reflects the number of times an identified security patch actually is security related. In other words, it indicates how precise the model is in identifying security patches. The formula of $Precision$ is shown as (3)

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

where TP is short for True Positive, which is the number of predicted security patches that actually are security related. FP refers to False Positive, which is the number of predicted security patches that actually are not.

$Recall$, aka True Positive Rate (TPR), is the measurement describing how robust the model is in identifying security patches. It can be calculated as (4)

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

where FN refers to False Negative, which is the number of cases that are predicted as non-security patches actually are security patches. It can be seen that a high $Recall$ means the method can detect most security patches.

$F_1\ score$ is the harmonic mean between $Precision$ and $Recall$. A higher $F_1\ score$ implies a lower false positive rate as well as a lower false negative rate. Mathematically, $F_1\ score$ can be expressed as (5)

$$F_1\ score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{5}$$

$Accuracy$ is the ratio of the number of correct predictions to the total number of input samples. Thus, a high $Accuracy$ means that the model performs well in distinguishing security patches from non-security ones. It can be formulated as (6).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{6}$$

where TN is the abbreviation of True Negative, which is the number non-security patches that are correctly classified.

Environment Settings. We implement the prototype system in Python 3.10, while the neural network is designed based on TensorFlow 2.11. Our system is carried out in the Ubuntu 22.04.1 LTS environment running on a computer equipped with an Intel® Core™ i9 CPU, 64GB RAM. We realize the neural network training by employing a CUDA-based parallel computing platform with a NVIDIA® GeForce RTX™ 3080 GPU. The batch size of the model training is set to be 32, while the learning rate is set to be 1×10^{-4}.

B. Evaluation Results

As shown in Table II, the total test accuracy of our model is 91.7%. The precision, recall, and F_1 score are 93.0%, 85.5%, and 89.1%, respectively. Moreover, the False Positive Rate (FPR) is only 4.3%. We also use Receiver Operating Characteristic (ROC) Curve to display the relation between FPR and TPR for various thresholds. The ROC Curve is shown in Fig. 4, where the area under curve (AUC) value is 94.86%.

TABLE II
MODEL PERFORMANCE

Precision	*Recall*	*F_1 score*	*Accuracy*
93.00%	85.5%	89.1%	91.7%

It is worth pointing out that the distribution of security patches and non-security ones is not balanced in practice. Most patches should be not be related to vulnerabilities fixing. They may be used for diverse purposes, such as functionality changes or performance improvements. Thus, the FPR is a very important measurement to evaluate the model performance: a lower FPR indicates that the system makes fewer mistakes for non-security patches. The existing works usually either neglect reporting a FPR value or underperform in achieving a low FPR. For instance, the reported FPR in [13] is 11.58%. Comparing with them, the proposed method is able to keep both a high detection rate and a very low FPR.

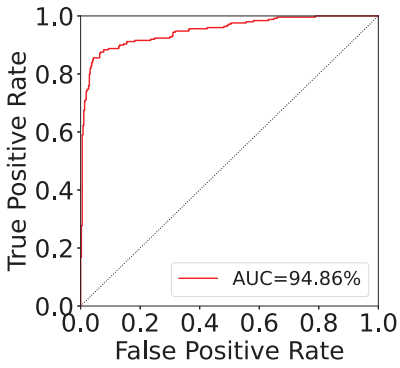

Fig. 4. ROC Curve.

C. Silent Security Patches Discovery

As a case study, we extend our experiments to two popular open-source projects (i.e., The PHP Interpreter and Radare2) to discover a number of secret security patches. We leverage our system to identify the security patches from the commits of the two aforementioned projects from GitHub. Once a security patch is found, we manually examine whether there exists a corresponding CVE record. Limited by our manpower, we eventually confirm 48 silently fixed vulnerabilities. Table III summarizes 20 security patches out of them due to space constraints, where half entries are from php-src repository, and others are from radare2 repository. The first column of the

TABLE III
SILENTLY FIXED VULNERABILITIES†

Source repository (commit hash)	*Commit date*	*Vulnerability type*
php-src(085af56710...73d)	05/17/2008	❶
php-src(a63d0f55da...59c)	01/25/2016	❷
php-src(0f20e113c2...182)	05/03/2016	❸
php-src(21452a5401...234)	10/11/2016	❹
php-src(4273865bba...396)	10/14/2019	❺
php-src(f79c774274...89f)	01/20/2020	❺
php-src(a681b12820...635)	03/30/2020	❶
php-src(a08847ab39...600)	03/17/2021	❺
php-src(45cb3f917a...cd5)	11/21/2022	❷
php-src(47ed1904ef...e53)	02/02/2023	❺
radare2(31e8d7cb68...559)	12/01/2016	❷
radare2(52826ff3ef...193)	08/26/2017	❷, ❸, ❺
radare2(6d4950b694...c9a)	08/14/2018	❺
radare2(e345e7d829...2b8)	03/03/2020	❷
radare2(053fca8585...c73)	05/13/2021	❷, ❸
radare2(d90510f951...87e)	05/21/2021	❷
radare2(c2fe231a6a...7f7)	05/30/2021	❸
radare2(f59b7dfb56...a85)	02/18/2022	❻
radare2(a2a29d8a24...858)	07/31/2022	❷
radare2(e87420bfc3...711)	02/24/2023	❷

† ❶ Integer overflow; ❷ Memory leak; ❸ Double free; ❹ Heap overflow; ❺ Use after free; ❻ Divide by zero.

table showcases the hash value used as the unique identifier of each patch commit in source repository. Each value in the commit date column, that is direcctly obtained from the patch, represents the vulnerability fix date. Finally, we annotate the vulnerability type for each entry indicated by the third column. It can be seen that the proposed system is beneficial to the detection of silent security patches.

V. RELATED WORK

In this section, we briefly review the recent advances in the pertinent field. But, given the large body of research on the patch analysis, we only cover the most related works.

Firstly, most of the existing work on patch classification extract statistic features from the source code of the patch. In detail, Perl et al. [23] propose many statistic differences between vulnerability related and other commits. Nevertheless, their concentration is not to distinguish vulnerability commits from non-security commits. Wang et al. [11] leverage 61 features to form an input vector in their machine learning based model, where 22 features are borrowed from [12]. In particular, the number of modified conditional statements, loops, lines, characters, and function calls are some sample features selected by both [11] and [12] to describe code changes. However, the feature engineering usually relies on heuristics or domain knowledge. For example, one heuristic adopted by [11] is "*security patches are more likely to modify less code than non-security patches*".

Secondly, some recent works utilize the natural language processing techniques to distinguish security patch from others. For example, Sabetta et al. [24] build a bag-of-words based classifier, which jointly analyzes both log messages and patches. As a result, they can achieve a precision of 80%. Similarly, Wang et al. [13] take both commit messages and patches into account. They thus construct a Recurrent Neural Network (RNN) based model. Finally, the total accuracy of their model is 83.57%, but the correpsonding false negative rate is also as high as 26.34%. In [14], the final prediction also comes from both code changes in patches and commit messages, which are analyzed using a BiLSTM and a graph neural network (GNN), respectively. Their experiments demonstrate a precision of 85.04%. In addition, Zhou et al. [15] choose a LSTM network to extract semantics from commit messages, while propose leveraging LSTM+CNN structure to learn code revision in patches. This hybrid model shows a precision of 86.24% in experiments. Although the proposed method in this paper also takes advantage of natural language processing technique to identify security patches, we mainly focus on commit messages and explore to what degree they can help in the classifier construction.

Thirdly, some empirical studies on bug reports have been done in previous works [25]–[27]. These researches share the similar research question, that is to explore whether security bugs are treated with a higher priority in practice. More concretely, the quantitative analysis in [26] and [27] are only carried out in Chromium browser project. In particular, when documenting a bug in the repository, the report follows a keywords-based labeling criteria. Consequently, both [26] and [27] adopt a straightforward method to classify the bugs, that is by searching the keywords 'security' and 'performance' directly in the bug reports. Besides, the case study in [25] merely involves the Firefox browser project. The bug classification is also completed based on a pattern matching strategy, where manual efforts and expertise are required for pre-defining a set of keywords. More importantly, since the researches are limited to specific software projects, extending their approaches to other open-source software is impractical.

VI. DISCUSSION

First of all, the foremost concentration of this study is to explore how much the textual description alone can impact on final security patch detection, therefore the proposed approach could be less helpful if the commits do not contain any textual information. However, the proposed method still can be utilized as a powerful building block and integrated into other existing security patch detection systems. More importantly, through this research we would like to highlight the importance of well-crafted commit messages and attract more attention from the open-source community.

In addition, we conduct our research merely based on the patch commits collected from GitHub, because currently it is the largest source code host. However, we do notice that there exist other platforms that provide Git repositories as a

service such as GitLab[2] and SourceForge[3]. Therefore, we plan to collect more data from diverse platforms in the next stage.

Last but not least, we have realized that when submitting a patch commit, many contributors would update not only the source code files but also the log files. The log files usually contain more detailed textual explanations about the updates. We do believe that mining the log files definitely would be an important supplement to the existing scheme. Thus, we leave this direction as a future work.

VII. Conclusion

In this paper, we propose a Transformer-based approach to automatically identify security patches, which is the first work that only focuses on textual descriptions in commits. To tackle the specificity of textual descriptions, we develop a regular expression based pre-processor. Then, we leverage the skip-gram model to generate a representation matrix as the input for the neural network. Finally, we apply the encoder component of the Transformer architecture to distinguish the security patches. The evaluation results on a real-world large-scale patch commit dataset collected from GitHub demonstrate the proposed method can achieve a high precision, i.e. 93.0%, with low false alarms. Our case study also discloses 48 silent security patches. More importantly, the findings from this work deepen our understanding of how important a high-quality commit message is for the later software maintenance. At last, we hope that through this research, we can appeal to the open source community for adequate emphases on the commit messages.

References

[1] CVE, "Published CVE Records," https://www.cve.org/About/Metrics, accessed: 2023-03-01.

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

[3] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *The 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 705–716.

[4] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[5] L. Luo, Q. Zeng, B. Yang, F. Zuo, and J. Wang, "Westworld: Fuzzing-assisted remote dynamic symbolic execution of smart apps on iot cloud platforms," in *Annual Computer Security Applications Conference*, 2021, pp. 982–995.

[6] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[7] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *Advances in Neural Information Processing Systems*, vol. 28, 2015.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[9] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[2]https://about.gitlab.com/

[3]https://sourceforge.net/

[10] F. Zuo and Q. Zeng, "Exploiting the sensitivity of L2 adversarial examples to erase-and-restore," in *The ACM Asia Conference on Computer and Communications Security*, 2021, pp. 40–51.

[11] X. Wang, K. Sun, A. Batcheller, and S. Jajodia, "Detecting 0-day vulnerability: An empirical study of secret security patch in OSS," in *The 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 485–492.

[12] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *The 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 386–396.

[13] X. Wang, S. Wang, P. Feng, K. Sun, S. Jajodia, S. Benchaaboun, and F. Geck, "Patchrnn: A deep learning-based system for security patch identification," in *2021 IEEE Military Communications Conference (MILCOM)*, 2021, pp. 595–600.

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

[15] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "SPI: Automated identification of security patches via commits," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–27, 2021.

[16] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *ICLR Workshop*, 2013.

[17] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in Neural Information Processing Systems*, vol. 26, 2013.

[18] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, 2015.

[20] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python*. O'Reilly Media, Inc., 2009.

[21] B. Sigurd, M. Eeg-Olofsson, and J. Van Weijer, "Word length, sentence length and frequency–zipf revisited," *Studia linguistica*, vol. 58, no. 1, pp. 37–52, 2004.

[22] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *The 7th international conference on Language Resources and Evaluation (LREC) Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, 2010, pp. 45–50.

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

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

[25] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on Firefox," in *The 8th Working Conference on Mining Software Repositories*, 2011, pp. 93–102.

[26] J. Imseis, C. Nachuma, S. Arifuzzaman, M. Zibran, and Z. A. Bhuiyan, "On the assessment of security and performance bugs in chromium open-source project," in *The 5th International Conference on Dependability in Sensor, Cloud, and Big Data Systems and Applications*, 2019, pp. 145–157.

[27] A. Rajbhandari, M. F. Zibran, and F. Z. Eishita, "Security versus performance bugs: How bugs are handled in the chromium project," in *The 20th IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (SERA)*, 2022, pp. 70–76.