

# PatchRNN: A Deep Learning-Based System for Security Patch Identification

Xinda Wang\*, Shu Wang\*, Pengbin Feng\*, Kun Sun\*, Sushil Jajodia\*, Sanae Bennaaboun<sup>†</sup>, and Frank Geck<sup>†</sup>

\*Center for Secure Information Systems, George Mason University, Fairfax, VA, USA

<sup>†</sup> CSIA Division, C5ISR Center, Space and Terrestrial Communications Directorate,  
U.S. Army Combat Capabilities Development Command (DEVCOM)

{xwang44, swang47, pfeng4, ksun3, jajodia}@gmu.edu, {sanae.m.bennaaboun.civ, frank.c.geck.civ}@mail.mil

**Abstract**—With the increasing usage of open-source software (OSS) components, vulnerabilities embedded within them are propagated to a huge number of underlying applications. In practice, the timely application of security patches in downstream software is challenging. The main reason is that such patches do not explicitly indicate their security impacts in the documentation, which would be difficult to recognize for software maintainers and users. However, attackers can still identify these “secret” security patches by analyzing the source code and generate corresponding exploits to compromise not only unpatched versions of the current software, but also other similar software packages that may contain the same vulnerability due to code cloning or similar design/implementation logic. Therefore, it is critical to identify these secret security patches to enable timely fixes. To this end, we propose a deep learning-based defense system called PatchRNN to automatically identify secret security patches in OSS. Besides considering descriptive keywords in the commit message (i.e., at the text level), we leverage both syntactic and semantic features at the source-code level. To evaluate the performance of our system, we apply it on a large-scale real-world patch dataset and conduct a case study on a popular open-source web server software - NGINX. Experimental results show that the PatchRNN can successfully detect secret security patches with a low false positive rate.

**Index Terms**—security patch, open source software, deep learning, vulnerability mitigation

## I. INTRODUCTION

With the wide adoption of Open Source Software (OSS), software maintainers are usually overwhelmed by a large number of patches including both security patches and non-security patches. Since applying patches would introduce system downtime and extra workload (e.g., changing code for new API), security patches should take precedence over non-security patches to be applied for avoiding the N-day attack. However, software vendors may not provide sufficient information in the changelog when releasing these patches. In such a case, for maintainers, manually identifying the existence of security patches is time-consuming, labor-intensive, and error-prone. Failing to timely apply a security patch of Apache Struts 2, Equifax experienced a data breach [11] and exposed millions of personal information. To solve these problems, we propose PatchRNN, a deep learning-based system to help automatically identify the security patches.

To alleviate human efforts, machine learning has been widely adopted in patch analysis since it can automatically

identify target code samples containing similar patterns with those in the training data [6], [7], [29]. Even though, machine learning still requires human experts to define a set of distinguishable features. In contrast, deep learning is able to automatically extract features and learn their importance from training samples [10]. While neural networks have not been employed in patch analysis, successful application of it in natural language processing suggests that it may be suitable for programming language processing where context is also important. Therefore, we adopt Recurrent Neural Networks (RNNs) that are effective in processing sequential and context-sensitive data [19].

For an OSS project maintained using Git, a patch (i.e., a commit) is composed of two parts: commit message and source code change. Previous patch analysis works are mainly at the text level, i.e., analyzing security-related keywords in the commit message [5], [17]. However, software maintainers may not explicitly and accurately specify the security impact due to limited security expertise, different human subjectivity, and changed maintenance policy [2]. In this case, we propose to consider information provided at the source-code level. Although some approaches also make use of source code in the patch, they extract the metadata like the number of lines, hunks, etc. Instead, we focus on the syntax and semantics of the source code itself.

In our work, we use Recurrent Neural Network (RNN) to extract the syntax and semantic level information from patches. We utilize the information from both the diff code and commit message to capture more comprehensive features. For the commit message, we use a TextRNN model to get the message vector. For the diff code, we use a twin RNN-based network with two identical sub-networks to obtain a code vector. After aggregating these two vectors, we can have the final prediction results by adopting a two-layer fully connected network. Our experimental results on a large-scale real-world patch dataset show that we can achieve a total accuracy of 83.57% with an F1 score of 0.747. At the same time, the fall-out rate (false positive rate) is 11.58% while the miss rate (false negative rate) is 26.34%.

To further evaluate the effectiveness of our system, we perform a case study on a popular open-source web server software - NGINX, and discover 10 commits that make security-

```

1 From f58c25069cf4a986fe17a80c5b38687e31feb539 Mon Sep
  17 00:00:00 2001
2 From: Sebastian Pipping <sebastian@pipping.org>
3 Date: Wed, 10 Oct 2018 14:49:51 +0200
4
5     ResetUri: Protect against NULL
6
7 diff --git a/src/UriCommon.c b/src/UriCommon.c
8 index 3775306..039beda 100644
9 --- a/src/UriCommon.c
10 +++ b/src/UriCommon.c
11 @@ -75,6 +75,9 @@
12
13 void URI_FUNC(ResetUri)(URI_TYPE(Uri) * uri) {
14 +   if (uri == NULL) {
15 +       return;
16 +   }
17   memset(uri, 0, sizeof(URI_TYPE(Uri)));
18 }
19 }

```

Listing 1: An example of security patch for NULL pointer dereference vulnerability (CVE-2018-19200).

```

1 commit ac367d7a2884aa150cdfc0495348fd886d3bd228
2 Author: Embedthis Software <dev@embedthis.com>
3 Date: Thu Nov 12 10:59:07 2015 -0800
4
5     FIX: don't try to catch SIGKILL
6
7 diff --git a/src/goahead.c b/src/goahead.c
8 index 6e6c806a..aa66d292 100644
9 --- a/src/goahead.c
10 +++ b/src/goahead.c
11 @@ -204,7 +204,6 @@ static void initPlatform()
12 {
13     #if ME_UNIX_LIKE
14         signal(SIGTERM, sigHandler);
15 -       signal(SIGKILL, sigHandler);
16     #ifdef SIGPIPE
17         signal(SIGPIPE, SIG_IGN);
18     #endif

```

Listing 2: An example of non-security patch in *GoAhead* software.

related changes but are not explicitly described in the official changelog. Among them, corresponding vulnerabilities of 5 patches are ranked as top dangerous software weaknesses [3].

## II. BACKGROUND

This section provides the definition and composition of software patches. We also describe the differences between security and non-security patches.

**Software patch.** A software patch is a set of code changes between two versions to address security vulnerabilities, resolve functionality bugs, or add new features. On a version control platform like GitHub [1], a commit can be regarded as a patch since it is good practice to separate updates for different issues. As shown in Listing 1 and 2, each commit is mainly composed of two parts: the commit message that describes the commit using the natural language and the source code difference. A set of consecutive removed and added statements (i.e., lines start with `-` or `+`) with their context lines is called one hunk. Besides, a 20-byte long hash string is used to uniquely identify a patch. A patch may have more than one hunk that modifies multiple files and multiple functions. A line starting with *diff* `--git` is used to point out the modified files.

**Security and non-security patch.** Security patches correct specific weaknesses described by a vulnerability. Non-security patches include bug fix patches and new feature patches. The bug fix patches make the software run more smoothly and reduce the likelihood of a crash by correcting the software bugs. The new feature patches add new or update existing functionality to the software.

Listing 1 is a security patch for vulnerability CVE-2018-19200 that prevents a NULL pointer dereference by adding a NULL check and corresponding handling (Line 15, 16, and 17) for `uri` whose pointed memory would be initialized as 0 (Line 18). Listing 2 shows an example of non-security patch. Since SIGKILL will cause the target process to terminate immediately, it cannot be intercepted. Thus, this patch avoids catching SIGKILL by removing the corresponding function call (Line 15).

## III. METHODOLOGY

### A. Overview

Figure 1 presents the architecture of our PatchRNN toolkit. Since a commit (i.e., a patch file) is composed of source code and commit message, we utilize both parts to capture more comprehensive features. For diff source code, we reconstruct the unpatched code and patched code and process them separately with the same tokenization and abstraction strategies. Then a twin RNN model is adopted to generate the code vector representation. For the commit message, we utilize the NLP toolkit to process the text sequences and employ a TextRNN model to obtain vector representation for commit message. The final results are derived from the feature vectors of both parts. We use a large-scale patch dataset *PatchDB* [22] to train the model. The size of the dataset is 38K.

### B. Feature Extraction

To identify the security patches, we focus on the syntax-level information in patches, which can be learned directly by recurrent neural networks. The overview of the patch identification scheme is shown in Fig. 2. In our scheme, the patch feature extraction contains two parts: diff code processing and commit message processing. We utilize the information from both the diff code and commit message to capture more comprehensive features. A diff file not only contains the diff code that can indicate the differences between unpatched code and patched code but also contains the commit message that may contain some clues to indicate if this patch is security-related. As a result, the commit message is also a critical component in our scheme.

1) *Feature Extraction from Diff Code:* First, we extract the diff source code from diff files and reconstruct the original unpatched code and patched code. Each source code will be concatenated into a sequence and will be separated into code tokens with the *Clang* tool [4]. The tokenization with Clang is a critical step since these tokens can become the direct inputs to a deep learning model. We find that all the tokenization tools in natural language processing (NLP) cannot work well for the programming language. For instance, the ‘equal to’

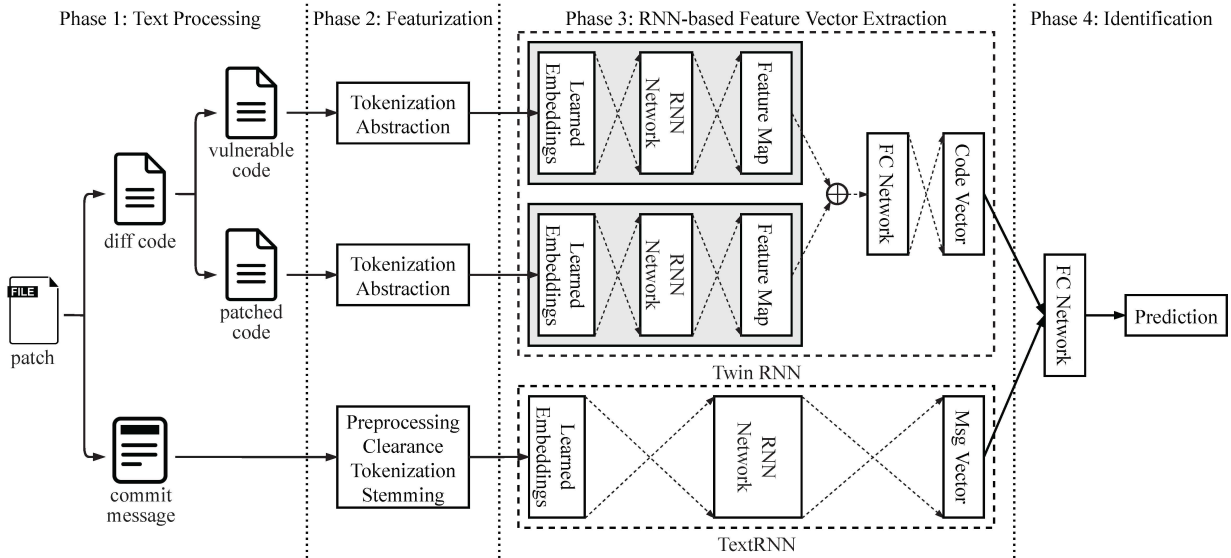


Fig. 1: The architecture of PatchRNN model.

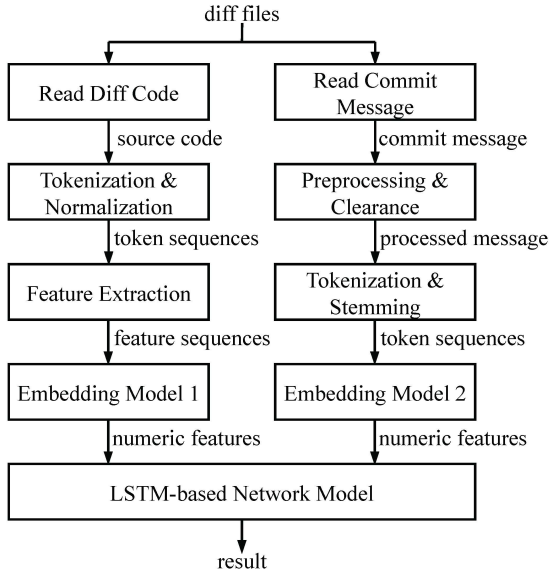


Fig. 2: The overview of patch identification scheme.

operator ('==') will be separated as two 'equal' signs ('=', '='), which do not conform to the meaning of programs. Instead, Clang is developed for C/C++ language thus is suitable for code tokenization. Moreover, for each token, we utilize Clang to extract additional features (e.g., the token type), which can better help us identify the security patches with syntax information. Except the special pad token (i.e., '<pad>'), we have five token types for normal tokens (i.e., *TokenKind.Keyword*, *TokenKind.Identifier*, *TokenKind.Literal*, *TokenKind.Punctuation*, and *TokenKind.Comment*). Another important feature is the diff type, which is a numeric feature that indicates if a token appears in a deleted line (−1), added line (+1), or a contextual line (0). In our design, the diff

type feature is suitable for the patch file to distinguish the vulnerable code and patched code.

Second, we abstract the code tokens according to their token types. The purpose of abstraction is to reduce the token variants and vocabulary size so as to achieve a better generalization ability of the deep learning model. As we know, some code tokens (e.g., variable names and function names) are defined by programmers thus have a lot of freedom in naming. It is inappropriate to assign each token with an individual embedding vector. Therefore, to handle the overfitting problem, we need to group the tokens with similar properties into one class. After the token abstraction, the vocabulary size of the dataset is reduced from over 600K to 28K. A example of token abstraction is illustrated in Fig. 3. We will remain the keywords and punctuation unchanged and only abstract the tokens of identifiers, literals, and comments. For the tokens of identifiers (i.e., variable names and function names), the abstracted tokens would be *VAR<sub>n</sub>* and *FUNC<sub>n</sub>*, respectively. For the tokens of literals, we further divide the literals into constants and strings. We only abstract the strings into a fixed token *LITERAL* but leave the constants unchanged, because some security issues (e.g., buffer overflow) are highly related to the index. For the tokens of comments, we delete them because the comments make a little contribution to our patch identification tasks.

Third, we normalize the token sequences to a fixed length. We analyze the cumulative distribution function (CDF) of the token sequence length and find most of the token sequences have a short length. We set the normalized sequence length as 1,100 tokens because it can cover 95% patch samples. If the sequence length is less than 1,100 tokens, we will pad a special token '<pad>' at the end of the sequence.

Finally, we convert each code token into an embedding vector by adopting an embedding model *word2vec* that utilizes the skip-gram and CBOW algorithm. The embedding vector is



```

if (snprintf(spath, sizeof(spath), var, iface)
    >= sizeof (spath))
    return -1;

+ /* No path traversal */
+ if (strstr(name, "..") || strchr(name, '/'))
+     return -1;
+
if (access(spatch, F_OK) != 0)
    return -1;

```

(a) original diff code.

```

if (FUNC0(VAR0, sizeof(VAR0), VAR1, VAR2) >=
    sizeof (VAR0))
    return -1;

+ if (FUNC1(VAR3, LITERAL) || FUNC2(VAR3, LITERAL))
+     return -1;
+
if (FUNC3(VAR0, VAR4) != 0)
    return -1;

```

(b) abstracted diff code.

Fig. 3: The token abstraction processing of diff code.

a learned dense representation for token features where tokens of the same meaning will have a similar representation. In our design, the dimension of code embeddings is set to be 128. All of the features above are the inputs of the twin RNN model.

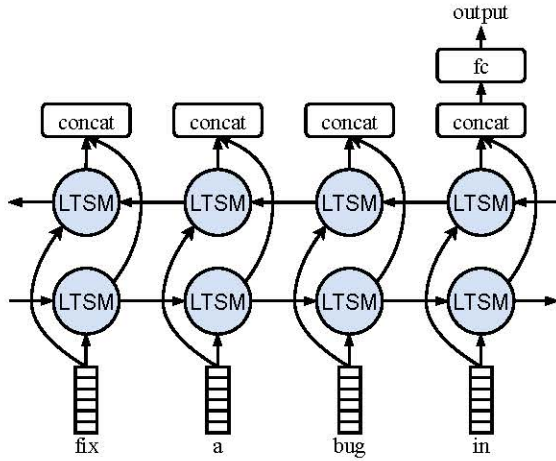


Fig. 4: The structure of recurrent neural network model.

2) *Feature Extraction from Commit Message*: The commit message in a patch contains modification information such as the reason for changing the code. For the commit message, we leverage a traditional natural language processing toolkit *NLTK 3.3* for text processing, including preprocessing, clearance, tokenization, and stemming.

First, all the letters in a commit message will be converted to lowercase. Second, we will clear the message with regular expressions to remove unimportant information, such as URL links, independent numbers, and the signatures in the footnote. Third, the processed commit message will be separated into a set of word tokens with a tokenization toolkit *TweetTokenizer*. We further clear the tokens without any English letter and the tokens of the email address. We also remove the tokens in the English *stopwords* list since these words cannot provide any unique information for security patch identification. Finally, a stemming tool *PorterStemmer* is utilized to stem the word tokens, reducing the inflected words or derived words to their base form. The reason that we stem the word tokens is to improve the generalization performance of the learned model

and reduce the vocabulary size. The sequence of word tokens will be normalized to the length of 200 words.

All the word tokens in the commit message are converted into 128-dimensional word embeddings via *word2vec* embedding model. After converting a commit message into a sequence of embedding vectors, we will feed it into a TextRNN model to obtain a message feature vector.

### C. Model Learning

We use an end-to-end deep learning model to convert a patch input to a prediction label. The learned model is based on RNNs that are suitable for sequential data. Given a patch, we extract the features from the diff code part and commit message part separately and feed them into the model. Our model contains 4 parts: twin RNN, TextRNN, feature fusion, final prediction.

For the diff code part, we use a twin RNN-based model to obtain the code vector. The inputs of the twin RNN are two sequences of feature vectors from unpatched code and patched code respectively. Each feature vector in the sequences is 135-dimensional, containing a token embedding, token type, and diff type. The token embedding is a 128-dimensional vector derived from an embedding layer. Token type feature is a 6-dimensional one-hot vector that indicates 5 Clang-defined token types and '<pad>'. The diff type feature is a numeric value that can be  $\pm 1$  or 0. In the twin RNN model, there are two LSTM-based sub-networks that share the same weights as well as the same functionality. That means the weights in the sub-networks are adjusted synchronously during model training. Each sub-network contains 2 bi-directional LSTM layers with a hidden layer size of 32. The outputs of two sub-networks will be concatenated and fed into a fully connected network with the dimension of [256, 128, 64]. Therefore, the output code vector is 64-dimensional.

For the commit message part, we use a TextRNN model to get the message vector. The structure of the RNN model is illustrated in Figure 4. A TextRNN model contains an embedding layer of 128 dimensions, a bi-directional LSTM layer of size 32. The LSTM outputs in the last position are concatenated as the input of a fully connected network with the dimension of [64, 64]. The message vector is hence a 64-dimensional vector.

Finally, the code vector and message vector are concatenated and fed into the prediction model, which contains a 3-layer fully connected network with the dimension of [128, 32, 2] and a softmax layer.

#### IV. EVALUATION

**Dataset.** We select *PatchDB* [22] as the dataset in our experiments, which is a large-scale patch dataset including both security and non-security patches in C/C++ extracted from the NVD and popular GitHub repositories. In total, there are 38,041 patch samples composed of 12,476 security patches and 25,565 non-security ones. Among them, we randomly choose 80% instances as the training set and the remaining 20% as the testing set.

**Setup.** Our program is implemented using Python 3.7, while the neural network model is designed based on PyTorch 1.6. Our model is carried out in the Ubuntu 20.04.1 LTS environment running in Intel Xeon Gold 5122, 3.60-GHz CPU with 64-GB RAM. We realize the neural network training by employing a CUDA-based parallel computing platform with 2 NVIDIA RTX 2080 Ti GPUs of 11 GB memory. The batch size of the model training is set to be 512, while the learning rate is set to be  $5 \times 10^{-4}$ .

	Actual Non-Security	Actual Security
Predicted Non-Security	4515 (T.N.)	659 (F.N.)
Predicted Security	591 (F.P.)	1843 (T.P.)

TABLE I: Confusion matrix of the predictions in our model.

**Performance.** In the experiments, 30K samples are used for training, 7,600 samples are used for testing. Our confusion matrix is shown in TABLE I. The total test accuracy of our model is 83.57% with the precision of 75.72% and the recall of 73.66%. The F1 score is 0.747. The fall-out rate (false positive rate) is 11.58% while the miss rate (false negative rate) is 26.34%.

**Overhead.** The time of preprocessing each sample in the dataset is 4.4 sec. For the neural network model, the training time is 27 min for 1,000 epochs. The training phase takes up 10.2GB of GPU memory. For a single patch, the prediction time is 5.6 sec on average including the preprocessing and resource loading.

#### V. CASE STUDY: NGINX

To further evaluate the performance of the PatchRNN, we conduct a case study on a popular open-source web server software - NGINX and find some secret security patches.

Although maintained on GitHub [13], the NGINX website [14] would also list the changes when a new version is released. These changes would be tagged as “security”, “bugfix”, “feature”, “change”, etc. However, we find that, even though no changes are labeled as security ones for some releases, some GitHub commits are for security fixes. Such commits can be regarded as secret security patches since vendors do not explicitly release their potential security

TABLE II: the comparison between the number of security issues listed in the documentation and the number of security patches committed on the GitHub repository.

Changes w/	Documentation (# Items)			Ground Truth (# Commits)	
	Security	Bugfix	Other	Security	Non-Security
1.19.1	0	6	5	8	11
1.19.2	0	7	2	8	7
1.19.3	0	5	4	7	12

TABLE III: Detection Results on NGINX.

Changes w/	Documentation Security Issues	Ground Truth Security Patches	Detection Results	
			T.P.	F.P.
1.19.1	0	8	4	0
1.19.2	0	8	3	0
1.19.3	0	7	3	0
Sum.	0	23	10	0

impacts, which are highly likely to be ignored by software maintainers or users.

To figure out the number of security patches in the NGINX, we apply our tool on its GitHub commits. Since we need to manually check each commit to get the ground truth, we focus on three new versions (i.e., NGINX 1.19.1, 1.19.2, and 1.19.3).

Table II shows the number of security, bug fix, and other changes with NGINX 1.19.1, 1.19.2, and 1.19.3. The official documentation does not mention any security changes in these versions. However, by manually checking the GitHub commits of these versions, we find there are 8, 8, and 7 commits that are security-related, respectively.

After applying our PatchRNN on commits of the above three versions, we summarize the detection results in Table III. In total, our toolkit identifies 10 security patches that are secretly released by NGINX. Among all security patches, 43% (10 out of 23) are successfully identified. By manually checking their corresponding vulnerability types, we find 5 out of them are ranked as one of the CWE top 25 most dangerous software weaknesses [3], e.g., use-after-free, NULL pointer dereference, out-of-bound access, etc. Meanwhile, the PatchRNN does not introduce any false positive cases. That is to say, no non-security patches are labeled as security-related. In such cases, once a security patch is identified, it can be directly prioritized to be applied and it is highly likely to be a real security patch.

#### VI. RELATED WORKS AND DISCUSSION

**Patch Analysis.** Most of the previous works on patch analysis focus on leveraging natural language description [6], [7], [29]. They extract textual information like keywords in the bug report, commit message, changelog, etc. However, these methods require consistent and well-maintained documentation [15], which performs limited generalization capability in newly released patches or patches for other projects. Some works make a step forward by retrieving and analyzing the source code part of the patch [12], [16], [18], [20], [21], [23], [28]. They study the attributes of metadata and syntax like the number of deleted lines, conditional statements, program inputs, etc.

However, these works do not distinguish between security patches and non-security bug fixes. Although some works [9], [26] perform empirical studies on differences between security and non-security patches, they do not provide a practical method to help automatically identify security patches.

At the binary level, Xu et al. [25] analyze the difference in execution traces to identify the existence of a security patch. Given the source code of a security patch, some researchers [8], [27] propose methods to test if the corresponding vulnerabilities have been patched in the binaries.

**Discussion.** Currently, the PatchRNN only supports C/C++ that are languages with the highest number of vulnerabilities [24]. Future work can extend it by applying other analysis tools to parse other programming languages. Also, the input of our system is not limited to GitHub commits. For projects that are not maintained by Git, its changelog can be used as the commit message, and code difference can be computed between two neighboring versions.

## VII. CONCLUSION

In this work, we propose the first deep learning-based approach that automatically identifies security patches to prioritize their application. We leverage both the commit message and source code difference to overcome the situation where the documentation of a patch is not well-maintained. Then, we apply TextRNN and twin RNN, respectively. The evaluation results on a real-world large-scale patch dataset and well-known web server software show that we could achieve good performance with low false alarms.

## VIII. ACKNOWLEDGMENTS

This work was partially supported by the US Department of the Army grant W56KGU-20-C-0008 and the National Science Foundation grant CNS-1822094.

## REFERENCES

- [1] GitHub. <https://github.com>.
- [2] The biggest and weirdest commits in linux kernel git history. [www.destroyallsoftware.com/blog/2017/the-biggest-and-weirdest-commits-in-linux-kernel-git-history](http://www.destroyallsoftware.com/blog/2017/the-biggest-and-weirdest-commits-in-linux-kernel-git-history), 2017.
- [3] 2020 CWE Top 25 Most Dangerous Software Weaknesses. [https://cwe.mitre.org/top25/archive/2020/2020\\_cwe\\_top25.html](https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html), 2020.
- [4] Clang: a C language family frontend for LLVM. <https://clang.llvm.org>, 2021.
- [5] Gabriele Bavota. Mining unstructured data in software repositories: Current and future trends. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 1–12. IEEE, 2016.
- [6] Dipok Chandra Das and Md Rayhanur Rahman. Security and performance bug reports identification with class-imbalance sampling and feature selection. In *2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*, pages 316–321. IEEE, 2018.
- [7] Katerina Goseva-Popstojanova and Jacob Tyo. Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 344–355. IEEE, 2018.
- [8] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zheming Yang. Pdiff: Semantic-based patch presence testing for downstream kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1149–1163, 2020.
- [9] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.
- [10] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.
- [11] Lily Hay Newman. Equifax officially has no excuse. <https://www.wired.com/story/equifax-breach-no-excuse/>, 2017.
- [12] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. Spider: Enabling fast patch propagation in related software repositories. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.
- [13] NGINX. <https://github.com/nginx/nginx>, 2021.
- [14] NGINX changes. <https://nginx.org/en/CHANGES>, 2021.
- [15] Mayana Pereira, Alok Kumar, and Scott Christiansen. Identifying security bug reports based solely on report titles and noisy data. In *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 39–44. IEEE, 2019.
- [16] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. VCCFinder: finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437. ACM, 2015.
- [17] Eddie A Santos and Abram Hindle. Judging a commit by its cover; or can a commit message predict build failure? *PeerJ PrePrints*, 4:e1771v1, 2016.
- [18] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 512–515. IEEE, 2016.
- [19] Jinsong Su, Zhixing Tan, Deyi Xiong, Rongrong Ji, Xiaodong Shi, and Yang Liu. Lattice-based recurrent neural network encoders for neural machine translation. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, AAAI’17*, page 3302–3308. AAAI Press, 2017.
- [20] Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. In *2012 34th international conference on software engineering (ICSE)*, pages 386–396. IEEE, 2012.
- [21] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. Detecting “0-day” vulnerability: An empirical study of secret security patch in oss. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 485–492. IEEE, 2019.
- [22] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. Patchdb: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 149–160. IEEE, 2021.
- [23] Xinda Wang, Shu Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. A machine learning approach to classify security patches into vulnerability types. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9. IEEE, 2020.
- [24] White Source Software. What are the most secure programming languages? <https://www.whitesourcesoftware.com/most-secure-programming-languages/>.
- [25] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering*, pages 462–472. IEEE Press, 2017.
- [26] Shahed Zaman, Bram Adams, and Ahmed E Hassan. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories*, pages 93–102, 2011.
- [27] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 887–902, 2018.
- [28] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 913–923. IEEE, 2015.
- [29] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 914–919, 2017.