A Machine Learning Approach to Classify Security Patches into Vulnerability Types

Xinda Wang[1], Shu Wang[1], Kun Sun[1], Archer Batcheller[2], Sushil Jajodia[1]
[1]*Center for Secure Information Systems, George Mason University, Fairfax, VA, USA*
[2]*Northrop Grumman, Washington, D.C., USA*
{xwang44, swang47, ksun3, jajodia}@gmu.edu, archer.batcheller@ngc.com

***Abstract*—With the increasing usage of open source software (OSS) in both free and proprietary applications, vulnerabilities embedded in OSS are also propagated to the underlying applications. It is critical to find security patches to fix these vulnerabilities, especially those essential to reduce security risk. Unfortunately, given a security patch, currently there does not exist a way to automatically recognize the vulnerability that is fixed. In this paper, we first conduct an empirical study on security patches by type (i.e., corresponding vulnerability type), using a large-scale dataset collected from the National Vulnerability Database (NVD). Based on analysis results, we develop a machine learning-based system to help identify the vulnerability type of a given security patch. The evaluation results show that our system achieves good performance.**

***Index Terms*—security patch, empirical study, software maintenance, open source software**

I. Introduction

Open source software (OSS) has been widely used in both free and proprietary applications. Black Duck reports that 96% of their scanned applications contain open source components, which account for 57% of the code base on average [1]. With the skyrocketing number of open source vulnerabilities [2], software maintainers have to deal with a bunch of security patches. Usually, security patches cannot be blindly applied in production systems where OSS components have been customized to meet their specific requirements [3]. Also, they may need to prioritize the application of certain security patches after identifying the corresponding vulnerability types.

However, it remains a challenge to identify the type of corresponding vulnerability for a given security patch. Security patches may not be well documented due to the subjectivity of software maintainers, limited security expertise, and changing regulations during the software life cycle [2]. Thus, there is a lack of necessary information on the type and impact of the fixed vulnerability. Existing research studies focus more on distinguishing security patches from non-security ones [4]–[6]. Nowadays, the types of security patches are decided mainly through manual analysis, due to the lack of in-depth understanding of security patches and an automatic classification solution [7], [8].

In this paper, we first conduct an empirical study on the security patches by type, using a large-scale security patch dataset collected from the National Vulnerability Database (NVD) and then develop a machine learning-based system to help automatically identify the (vulnerability) type of a given security patch. Since there is no available large-scale security patch dataset, we build one by querying the vulnerabilities in the NVD and then downloading their corresponding security patches. To guarantee the quality of our dataset, we manually go through all collected security patches and remove those mingled with other non-security ones [9]. We focus on the security patches of ten common vulnerability types in C/C++ languages [10], [11].

Comparing to researches that treat all types of security patches equally [4]–[6], [12], [13], our work is the first to explore the nature of security patches by type and reveal several new discoveries: (1) Many security patches do not localize in one function as expected by previous researches [14], [15]; (2) Some types of security patches such as *information exposure* and *path traversal* only make slight changes even if involving multiple functions; (3) Most vulnerabilities are patched by adding or updating conditional statements. All these observations provide useful insights to the development and improvement of vulnerability mitigation approaches.

We extract features inspired by our quantitative analysis and apply them in developing a machine learning-based security patch classification system. Compared with previous works that require well-documented metadata and is restricted to a small number of projects (most limited to only one project) [16], [17], our model uses source-code level features and shows better performance even on hundreds of projects. Our proposed method can be adopted by existing vulnerability tracking systems as a general component to reduce human efforts and biases on vulnerability classification.

In summary, we make the following contributions:

- We present a security patch classification system that can identify the vulnerability types of given security patches from multiple OSS projects. By capturing the effective source code level features and using machine learning approaches, our system can provide better performance than previous works that focus on analyzing one project. Our approach is suitable to be integrated into current software maintenance systems.
- We propose a reproducible methodology to collect a large-scale security patch dataset with corresponding vulnerability type labels from the National Vulnerability Database.

This work was partially supported by the National Science Foundation grant IIP-1266147 and the US Department of the Army grant W56KGU-20-C-0008.

We manually clean the dataset to assure its quality.

- To the best of our knowledge, we are the first one to conduct extensive quantitative analysis on security patches by type. Our findings expose the limitations of prior approaches for general security patches. We believe our explorations into specific characteristics of different security patches would be useful for improving current vulnerability mitigation techniques.

II. Related Work

A. Security Patch Collection

Since a security patch aggregates both vulnerable code and the corresponding modification at the same time, many vulnerability detection researches construct security patch datasets. Kim et al. [18] acquire security patches from eight well-known Git repositories. Z. Li et al. [19] built a Vulnerability Patch Database (VPD) that consists of 19 products. However, the size of these datasets is not sufficient to perform a machine learning-based study and may introduce biases to analysis results. Considering thousands of CVE records on open source projects, F. Li et al. [5] build a large-scale security patch database by querying the NVD. However, they have not open-sourced their database to the public.

B. Patch Analysis

There have been many works on patch analysis. On the source code level, Zhong et al. [12] conduct an empirical study on bug fixes from six popular JAVA projects. Soto et al. [13] focus on patterns, replacements, deletions, and additions of bug fixes. Perl et al. [20] study the attributes of commits that lead to vulnerabilities. However, they do not distinguish security patches from bug fixes. Zaman et al. [4] discover the differences between security patches and performance bugs on a single project - Mozilla Firefox. Li et al. [5] are the first one to perform a large-scale empirical study of security patches versus non-security bug fixes, discussing the basic characteristics and life cycles of security patches. On the binary level, Xu et al. [6] present a scalable approach to identify the existence of a security patch through semantic analysis of execution traces. With the help of signatures generated from open-source patches, Zhang et al. [21] propose a precise and accurate method to test if the software has been patched. Until now, most of patch analysis researches characterize various types of security patches as a whole and do not distinguish among them. To the best of our knowledge, our paper is the first one to analyze security patches by type on a large-scale cross-repository dataset.

C. Classification of Software Metadata

There has been a line of works investigating the software metadata, which does not require the retrieval and analysis of the source code. Arya et al. [22] OSS issue discussions and propose an automatic approach to detect the types of discussion comments. Hindle et al. [17] make use of the keywords in the commit message (similar to change log), the author, and module/file type information to classify code changes into a specific maintenance category. Aghaei et al. [23] discover the relationship between CVE description information and CWE definition to do the mapping. CVE Details website [24] use the keywords of CVE descriptions to classify the vulnerabilities. However, results of all such techniques may be threatened by inaccurate commit messages or descriptions. A study has shown that over 40% of the vulnerability reports or CVE summaries contain inconsistent information [8]. Also, considering the reputation, software vendors may manipulate the commit messages or release notes to conceal serious problems [25]. Instead of using this descriptive information, our work focuses on analyzing the source code of security patches.

```
 1 diff --git a/pppd/options.c b/pppd/options.c
 2 index 45fa742..e9042d1 100644
 3 --- a/pppd/options.c
 4 +++ b/pppd/options.c
 5 @@ -1289,9 +1289,10 @@ getword(f, word, newlinep, fname
        )
 6      /*
 7       * Store resulting character for escape sequence.
 8       */
 9 -    if (len < MAXWORDLEN-1)
10 +    if (len < MAXWORDLEN) {
11          word[len] = value;
12 -    ++len;
13 +        ++len;
14 +    }
15
16      if (!got)
17          c = getc(f);
18 @@ -1329,9 +1330,10 @@ getword(f, word, newlinep, fname
        )
19      /*
20       * Ordinary char: store it in word and get another.
21       */
22 -    if (len < MAXWORDLEN-1)
23 +    if (len < MAXWORDLEN) {
24          word[len] = c;
25 -    ++len;
26 +        ++len;
27 +    }
28
29      c = getc(f);
```

Listing 1. A patch example (CVE-2014-3158).

III. Collection of Security Patch Database

This section introduces our efforts on collecting a large-scale security patch dataset and improving the data quality. We focus on ten types of common vulnerabilities by considering the categorization and frequency of the CWE top 25 dangerous software errors, which cover 90% of our collected dataset.

A. Preliminary

A patch consists of the differences between the old and the new version of files and some context information. Listing 1 is an example of the patch for CVE-2014-3158. Each patch may modify several files and all the modification on a file is called difference that starts with a *diff* (e.g., line 1). Each difference may contain multiple change hunks that include consecutive lines before and after modification with $-$ and $+$, respectively. For instance, lines 9 and 10 is a change hunk, and there are four hunks in this patch. On software development platforms like GitHub, a commit can be regarded as a patch.

B. Data Collection

To avoid biased conclusions drawn from a small number of software projects, we collect a dataset by first querying the NVD that covers numerous cyber security products and services. Since CVE ID is used as the index of NVD, we download the CVE list that records all the up-to-date assigned CVE IDs as the reference to query the NVD. In the CVE list, each CVE ID is followed with pertinent reference links of reports, advisories, and patches (if any). Based on our observation, we find the URLs that contain security patches have some uniform patterns. For the repositories hosted on GitHub, the patch URLs are like: *github.com/{owner}/{repo}/commit/{commit}*. By matching such a pattern in the reference links, we can download corresponding security patches of NVD entries.

To get the vulnerability type, we crawl the corresponding NVD page where each vulnerability type is represented with a Common Weakness Enumeration Specification (CWE) ID [26]. Since patches written in different programming languages may have different syntactic patterns, it is hard to cover the syntax of all types of languages. We focus on patches of projects written in C/C++ that are the languages with the highest number of vulnerabilities [2].

C. Data Cleaning

The collected security patches may be polluted in two ways: (1) A security patch mingles with other patches; and (2) A security patch makes key modifications on the non-C/C++ part.

Theoretically, a patch is corresponding to a vulnerability fix, non-security bug fix, or feature update. However, due to different version control philosophies, some software vendors may release a big "patch" that mingles multiple security patches or even multiple types of patches. In case that these patches threaten our analysis results, we manually filter them out by analyzing the commit message (if any) and the code of each patch in our dataset.

Besides, after excluding all the non-C/C++ projects, we find even the patches of C/C++ projects may contain modifications on non-C/C++ files such as *.changelog, .kconfig*, etc. Since we find most of non-C/C++ parts only make simple documentation modifications (e.g., version number or release note), we could simply remove these parts from corresponding patches. Through manual check, we find less than 1% security patches embed key modification (e.g., *.S* files to solve the dependency problem). We do not take them into consideration.

TABLE I
Distribution of Vulnerability Type

No.	Vulnerability Type	Support
1	Buffer error	1462
2	Resource management error	566
3	Improper input validation	447
4	Numeric error	394
5	Broken access control	294
6	Information exposure	262
7	Path traversal	55
8	Cryptographic issue	54
9	Injection	43
10	Improper authentication	18

D. Vulnerability Type

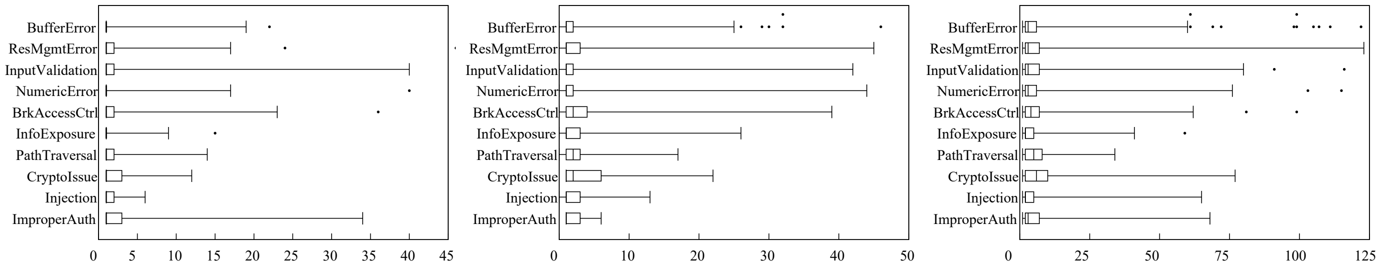
There are 67 CWE types in our original dataset. Here we do not directly adopt these CWEs as the ground truth of vulnerability type for our work. That is because the CWEs are organized in a hierarchical structure (tree) [26] where a CWE may be a subset (child) of other CWEs, facilitating the vulnerability evaluation on different granularity levels. However, it also introduces ambiguity and overlapping when CWEs of different granularity are used to label similar CVE entries [7]. For instance, CVE-2018-16392 that fixes out-of-bounds writes is labeled by an NVD analyst as CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer). Meanwhile, a similar vulnerability, CVE-2016-6855, is accurately labeled as CWE-787 (Out-of-bounds Write) that is the child of CWE-787. This is because NVD uses human analysts to manually score CVEs using CWEs and thus different analysts may have different opinions. Therefore, if we borrow CWE slice as vulnerability types, security patches for CWE-787 and CWE-119 would be regarded as two different classes while they are overlapping and have similar characteristics. Instead, we choose to group them into ten major types. First, we identify these types through considering both CWE top 25 dangerous software errors [10] and the frequency of each CWE type. Next, we study the definition of all the CWEs in our original dataset and conduct the remapping from CWEs to our identified ten major types. For instance, we regard CWE-787 and CWE-119 as a new type - *buffer error*. The above steps adopt a qualitative content analysis process as follows:

a. Two authors of this paper independently identify ten major types using their expertise after researching the CWE description and focusing on types with high frequency and severity. Then, they discuss their identified results, resolve disagreements (if any), and refine the major types.
b. Given the identified ten major types, two authors separately perform the remapping from 67 CWEs to ten major vulnerability types by studying the CWE definition. Then, they meet to achieve agreements on different results.

Table I demonstrates the distribution of ten major vulnerability types through the above analysis process. A brief description of each type is listed as follows:

Buffer error happens when software reads or writes a memory location that is outside of the intended boundary of the buffer and associated with other variables, data structures, or internal program data. This group includes buffer copy without input size checking (i.e., classical buffer overflow), out-of-bounds read/write, and access of uninitialized pointer. It may result in arbitrary code execution, control flow alteration, sensitive information exposure, or system crash.

Resource management error is composed of uncontrolled resource consumption, improper resource shutdown or release, race condition, double free, use after free, etc. It may result in the modification of unexpected memory locations, arbitrary



(a) File Number (b) Function Number (c) Hunk Number

Fig. 1. The patch localization at the file/function/hunk levels

code execution, or crash.

Improper input validation exists in the software when input is not validated properly. As a result, an attacker could manipulate the input to make it an unexpected form. It will lead to control/data flow alteration or arbitrary code execution.

Numeric error refers to integer underflow/overflow, incorrect conversion between numeric types, improper validation of array index, comparison without correct check, etc. It can cause sensitive content exposure or arbitrary code execution.

Broken access control includes improper access control and permission issues like missing authentication for critical function, use of hard-coded credentials, etc. This allows attackers to access unauthorized functionality or data, such as gaining privileges, accessing sensitive information, executing commands, evading detection, etc.

Information exposure consists of intentional and unintentional disclosure of information to some subjects that are not authorized to access. The severity mainly depends on the exposed information itself, e.g., financial data could be used for fraud.

Path traversal includes pathname traversal and link following that refer to the manipulation of special elements (such as ../ separators) of a path to escape outside of the restricted directory or access an unintended resource.

Cryptography issue covers various kinds of improper use of cryptography like missing or inadequate encryption of sensitive data, use of a broken or risky cryptographic algorithm, key management errors, etc.

Injection comprises OS command injection, cross-site scripting (XSS), SQL injection, and code injection.

Improper authentication occurs when the software incorrectly or improperly configures user and session authentication. It may allow attackers to compromise sensitive information like password, take control of users' accounts, etc.

IV. Security Patch Analysis

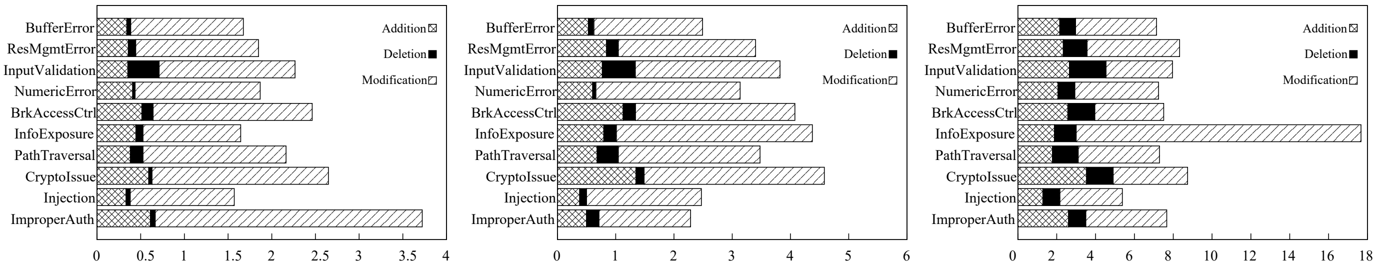
Instead of distinguishing security patches from non-security ones [4], [5], we aim to further classify among different types of security patches, since specific types of security patches should be applied with a higher priority than others in real-world scenarios. We characterize different types of security patches from three aspects: patch localization, patch complexity, and patch flow changes.

A. Patch Localization

Patch localization is an important characteristic that has been leveraged by many vulnerability related research. Some fault detecting approaches make the assumption that each buggy file contains exactly one line of faulty code [14]. Current automatic patching tools mainly focus on function-level patching with the assumption that vulnerable code is within one function [5], [15]. Some patch presence tools only deal with intra-procedural problems [6]. However, a vulnerability may locate in multiple functions with multiple non-aggregated lines that have dependencies among them, which limits the use of these techniques. In this paper, we analyze the patch localization for each type and our results provide new insights on the future research direction of locating and patching vulnerabilities.

We calculate the number of modified files, functions, and hunks for each type of security patches. Figure 1 shows the distribution of them and there are some new observations:

- Most of *buffer error*, *numeric error*, and *information exposure* vulnerabilities could be fixed within one file as shown in Figure 1(a), since similar functionalities are more likely to be aggregated in the same file. In contrast, applying patches for *cryptographic issues* and *authentication errors* requires careful analysis to avoid side effects on functionalities located in different files.
- 7 out of 10 vulnerability types have at least a quarter of security patches involving two or more functions (see Figure 1(b)), which is contrary to previous claims that most security patches only need to fix one function [5], [15]. Moreover, over half of security patches for *broken access*, *path traversal*, and *cryptographic issues* need to modify multiple functions. Therefore, for the system that bundles cryptography libraries or enforces access control mechanisms, previous works may raise non-negligible false negatives since inter-function vulnerabilities are totally ignored.
- Vulnerable code is usually not aggregated lines. That is because only consecutive modified lines would be a hunk and most security patches consist of multiple hunks (see Figure 1(c)). For human experts, patching such vulnerabilities requires a better understanding of context. As exceptions, the security patches of *information exposure* and *injection* vulnerabilities are more likely to contain



(a) File Number (b) Function Number (c) Hunk Number

Fig. 2. Number of changes at the file/function/hunk levels

only one hunk, since those vulnerabilities can be easily fixed by adding a sanity check once they are located.

Moreover, we use the security fix entropy [27] to quantify the localization of different security patches:

$$H_n(P) = -\sum_{k=1}^{n}(p_k \cdot \log_n p_k),$$

where $p_k > 0$ $(1 \leq k \leq n)$ and $\sum_{k=1}^{n} p_k = 1$. For instance, if a security patch modifies 3 functions with 1, 2, and 3 hunks, respectively, the p_1, p_2, and p_3 would be 1/6, 1/3, and 1/2. If the probability of modifying a specific function is 1 and all others are 0, the entropy will be minimal and means the fix is most localized. If multiple functions are involved in a patch and the number of hunks modified in each file is the same, the entropy will be maximal and this security patch is least localized. This metric is different from simply counting the numbers of functions and hunks. Considering two security patches where *A* modifies three functions with one, one, and four hunks and *B* modifies three functions with two, two, and two hunks, the latter two metrics evaluate them as the same since their number of functions and hunks are equal. However, patch *A* aggregates on one of three functions, which is less localized than patch *B* that modifies three functions equally.

Figure 3 (on next page) depicts the cumulative distribution functions (CDFs) of Shannon Entropy for each type of security patches. For 7 out of 10 vulnerability types, most security code changes are localized within one function or aggregated in one out of multiple functions since over 50% of them have a zero or small entropy. In contrast, patches for *broken access control*, *path traversal*, and *cryptographic issues* are more likely to involve multiple functions with evenly distributed modifications. For vulnerability detection techniques, in the former cases, it would be easy to first locate the main cause of the fault and then track the potentially effected part [28]. In the latter cases, the vulnerabilities are usually generated by the interaction among several seemingly benign parts, which is harder to be detected.

B. Patch Complexity

After locating the vulnerable code, automatic program repair (APR) techniques use search-based or brute-force approaches to generate candidates until a candidate passes the whole test suites [29], [30]. For vulnerabilities with high complexity, the search space of the general solution is large, which leads to the generation of an explosive number of possible fixes. To avoid this, it is necessary to recognize the vulnerabilities whose patches are likely to be complex and develop more suitable techniques for them [31]. Besides, the complexity of general patches has been widely used to assist code audits. VCCFinder discovers that vulnerability contributing commits make more additions, deletions, and modifications on average compared with other commits [20]. Patch complexity has also been adopted as features to distinguish bug or even security fixes from other patches [25], [32].

To quantify the security patch complexity, we calculate the number of deletion, addition, and modification at file, function, and hunk levels. The average results are shown in Figure 2. The deletion/addition refers to the whole hunk, function, or file that only contains deleted/added lines. The modification (i.e., update) is counted when it includes both deleted and added lines. At the file level (Figure 2(a)), fixing *improper input validation* is more likely to delete files. The number of deleted and added files are similar, which means most *input validation* patches tend to rewrite files instead of modifying existing files. On the contrary, repairing *numeric errors* and *cryptographic issues* usually does not need to delete files. In such cases, although the numbers of added files for most vulnerabilities are similar, with less deleted files, patches for *numeric errors* and *cryptographic issues* are more likely to add "real" new files (not the rewritten ones). At the function level (Figure 2(b)), besides the *improper input validation*, patches for *path traversal* replace many vulnerable functions with rewritten ones. Patching *buffer errors*, *numeric errors*, *cryptographic issues*, and *injection* mainly depends on adding brand new functions and modifying existing functions. At the hunk level (Figure 2(c)), there are no significant differences among most types of security patches. An exception is patching *information exposure* requires a tripled number of modifications than others, while with a similar number of added and deleted hunks.

We also choose lines of code (LOC) and characters of code as metrics in our scenario, as done in previous works [4], [33]–[35]. We treat the lines with $-$ in the patch as deletion and lines with $+$ as addition. The character number is the sum of characters in corresponding lines. Figure 5 presents the average number of lines and character changes. The results drawn from the left and right figures are consistent for most kinds of vulnerability fixes. Note that the number of deletion

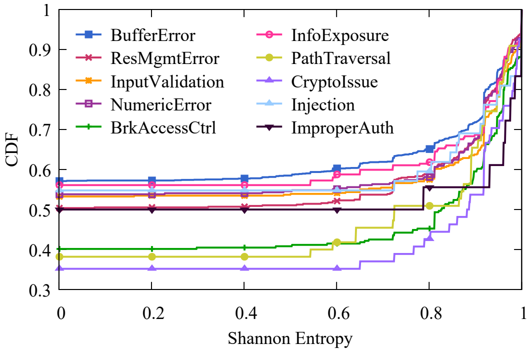


Fig. 3. CDFs of entropy

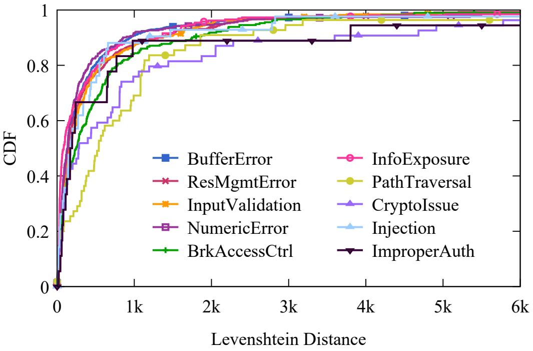
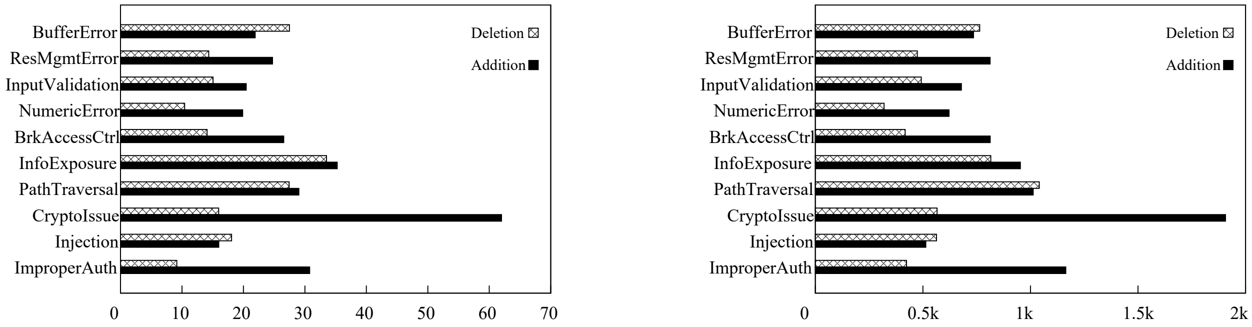


Fig. 4. Levenshtein distance of security patches



(a) Line Number

(b) Character Number

Fig. 5. Average numbers of line and character changes

changes for *cryptographic issues* is at the medium level while the largest number of additions are required. Similarly, the net number of addition (i.e., addition minus deletion) for *improper authentication* is also large due to a small deletion number and a medium addition number.

We find that the number of deletion and addition are very close at both line and character levels for *information exposure*, *path traversal*, and *injection*; however, we cannot say their modifications are slight since we do not know if the contents of deleted and added lines are similar. Since different programmers may have various styles of programming when implementing the software functionalities, it may not be accurate to only count the number of added and deleted lines. Also, it is unfair to equal a patch with ten completely modified lines with another patch with slight changes (e.g., operator) on ten lines. Therefore, we further quantify the patch complexity using the Levenshtein Distance [36], which is the total character number of deletions, insertions, and substitutions required to transform the vulnerable code to patched one. Since these operations are common and basic in security patches, Levenshtein Distance is intuitively a good criterion to measure the patch complexity. For instance, if a security patch changes $>$ to $>=$, then its Levenshtein Distance should be one. Figure 4 shows the CDFs of Levenshtein Distance for each type of security patches. For most vulnerability types, 80% of their security patches contain small modifications. The exceptions are *path traversal*, *cryptographic issue*, and *improper authentication* that require a large number of modifications. When using the LOC as metrics, security patches for *information exposure*, *path traversal*, and *injection* seem not complex. Using Levenshtein Distance, we can find that the patch complex for *information exposure* and *path traversal* are still simple. In other words, though their fixes involve many lines, corresponding modifications are small. In contrast, security patches for *injection* make more intensive modifications to existing code.

C. Patch Flow

It is critical to study the control flow changes in security patches since at least 60% security patches add or update conditional statements [37]. Here we aim to figure out what kinds of security patches are more likely to introduce certain types of control flow changes [6].

To quantify the flow changes in each type of security patches, we consider the conditional, loop, and jump statements. We implement a lexer and the detection results are presented in Figure 6. Usually, control flow will be modified after patching. The most prevalent flow changes are caused by conditional statements and the least one is due to jump statements. Fixing *buffer errors*, *improper input validation*, *numeric errors*, *broken access*, *path traversal*, and *injection* usually does not introduce new conditional statements. Based on our observation, the modifications of them are likely to be made on improving existing sanity checks (e.g., add or change some conditions). *Resource management errors* are mainly fixed by adding more conditional statements to limit the use of resources. Security patches for *information exposure* involve much more control flow statements compared with others. The *cryptographic issues* contain few loops and jumps in their vulnerable code

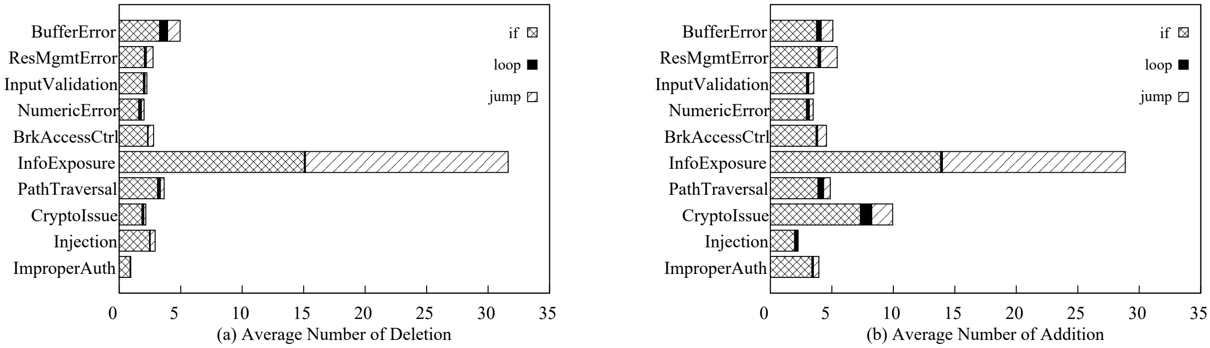


Fig. 6. Flow changes in deletion and addition of security patches

and add a lot when patching, which is chiefly imported from the newly added functions. *Improper authentication* commonly results from a lack of proper sanity checking so that many conditional statements are added in the patched version.

V. Security Patch Classification

To help understand the vulnerabilities fixed in given patches, we develop a machine learning-based system to classify security patches into corresponding vulnerability types, utilizing the varying characteristics of different types of security patches.

A. Feature Selection

We select a set of features from the metrics introduced in the last section plus the keyword features to characterize the security patches.

- *Patch Localization Features* consist of the number of modified files, the number of modified functions, the number of hunks, and Shannon Entropy of a security patch.
- *Patch Complexity Features* include the number of deletion, addition, and modification at file, function, and hunk level (addition/deletion refers to part with only added/deleted code, modification (i.e., update) that contains both deleted and added part); the number of lines and characters in addition (lines with $+$) and deletion (lines with $-$); and Levenshtein Distance between deletion and addition.
- *Patch Flow Features* comprise the number of condition, loop, and jump statements in addition and deletion.
- *Keyword Features* consider the number of each one in the C/C++ keyword list and library function list. We also take into account some "keywords" (e.g., *len*, *uid*, etc.) that are project-independent but may have strong correlations with specific kinds of security patches. Since there are thousands of features in this category, we rank these features with Mutual Information and choose a set of most useful ones.

B. System Modeling

Recall that Table I shows that our datasets are composed of ten imbalanced classes of security patches. To achieve the best performance, we apply multiple classification algorithms and balancing techniques and finally adopt Random Forest classifier with Synthetic Minority Over-Sampling Technique (SMOTE) and Class Balancer. We also conduct experiments with other classifiers (e.g., Support Vector Machine (SVM), Logistic Regression, and etc.), but their performances are not satisfactory and/or incur too long training time.

Random Forest is an ensemble learning method that is built up with multiple decision trees, where the leaves represent the predicted labels and the branches represent the decision conditions of multiple features. Random forest utilizes feature bagging that randomly selects a subset of features at each candidate split and can handle the data with relatively high dimensional features without feature selection.

To solve the imbalance problem, we use the synthetic minority over-sampling technique *SMOTE* [38] to re-sample the dataset. SMOTE synthesizes new samples for the minority class by randomly selecting a point in the line segment between a given sample and one of its k nearest neighbors in the feature space. Synthetic samples may provide additional information for classification but also introduce extra noise. The performance of the classifier may become worse when the oversampling rate is too high. Instead, we use *Class Balancer* to re-weight the instances of some classes to make sure each class will have the same total weights or reduce the weight difference among classes. When the weight of a class increases, misclassification of its instance will lead to more penalization, which raises the importance of minority class to avoid only predicting the majority class.

C. System Evaluation

We combine the Random Forest model with different balancing techniques and conduct 10-fold cross validation. Experimental results are shown in Table II. We see that Random Forest with Class Balancer presents the best results - 55% accuracy, i.e., given a security patch, our model could successfully classify it into one of ten classes with 55% possibility.

Since we are the first to classify the security patches into the corresponding vulnerability types, our experimental results cannot be directly compared with other works. Actually, there are some works on classifying commits (no matter security or non-security patches) into maintenance categories [16], [17]. We can have a rough comparison with those similar works to help evaluate our results. Hindle et al.'s work [17] is conducted

TABLE II
PERFORMANCE OF ML MODELS ON TESTING DATASET

ML Models	Accuracy	F1 Score
Random Forest	51.63%	0.455
Random Forest + SMOTE	54.54%	0.485
Random Forest + Class Balancer	54.75%	0.501

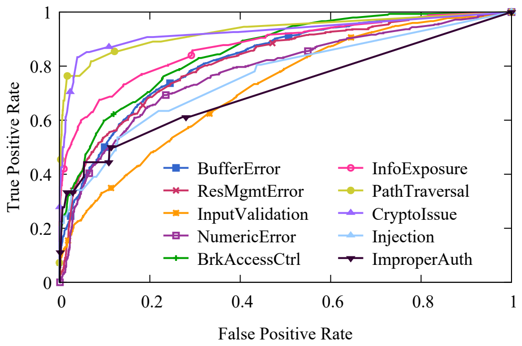


Fig. 7. ROC curve for Random Forest with Class Balancer

on single repositories and the accuracy is around 50%. Levin et al. [16] conduct experiments on a dataset that is composed of only nine projects and the maximum accuracy is 54%. It is clear the difference within different kinds of security patches is smaller than that among code changes of different maintenance activities, and keeping the model's generalization in the large-scale dataset with hundreds of projects is harder than several or even single repositories. Thus, our task is more complex than previous works. Besides, Hindle et al.'s solution is limited to code changes with well-maintained commit messages (i.e., change log), while our approach applies to all security patches since all the features are extracted from source code only. Also, although our dataset is highly imbalanced, our model does not simply predict any instances into the majority class (i.e., *buffer error*). Figure 7 shows that the area under the ROC curve (AUC) of each vulnerability type. The AUCs of *cryptographic error*, *path traversal*, *information exposure*, and *broken access control* are 0.928, 0.926, 0.873, and 0.852 that are larger than *buffer error* (0.823) and all close to one. The AUCs of other types are also acceptable. Therefore, our model is able to deal with various types of security patches with good performance. Also, we argue that the performance of our classifier could be further improved by better-defined vulnerability types. Currently, our vulnerability categorization is built on the top of CWE slices so that we can adopt CWE types of security patches for vulnerabilities in NVD as ground truth. But there may be some ambiguity and overlap among current types, e.g., *buffer error*, *improper input validation*, and *code injection*. In this case, our model may fail to distinguish among them. If NVD provides a better-defined vulnerability categorization, our results could be further improved.

VI. DISCUSSION

We discuss different scenarios where our system could be useful and describe some limitations as well as future work.

A. Usability

The security changes of large software are usually not well documented due to different analysis results from different maintainers and limited efforts to handle too many code changes throughout the life cycle. Our system could be integrated into the software maintenance system like GitHub to help eliminate the biases and save human efforts.

Our system is useful for developers whose software bundles multiple third-party libraries. It is well-known that it is hard to timely patch all known vulnerabilities in practice, since (i) some patches are not explicitly marked as a security fix and do not provide any vulnerability type information and (ii) customization on the source code makes security patches hard to be directly applied. Under such circumstances, our system can help identify the security patch types and prioritize the applications of specific security patches.

Our method could also be extended for the vulnerability tracking system like NVD where security patches are presented in each vulnerability entry (if available). NVD adopts Common Weakness Enumeration (CWE) IDs to represent vulnerability types and its categorization varies a lot each year. For instance, NVD embraces a new CWE slice in 2019 but only updates all the vulnerabilities after 2018 since analyst efforts are not enough to manage the huge number of vulnerabilities. But since vulnerability tracking systems feed valuable information to researches and practices, inconsistency between new and old vulnerability entries should be avoided. Our system could be extended to employ security patches with new CWE IDs when training the model. After that, the model is able to label security patches for outdated entries with new CWE IDs so that the vulnerability type for outdated entries will be updated.

B. Limitations

There are some limitations in our current work. The dataset collected from NVD may still be biased to some specific types of vulnerabilities (severe or highly exploitable ones). We argue that since the CVE list covers numerous cyber security products all over the world, our model could be applicable to most security patches. Also, even though our current model is in favor of severe or highly exploitable vulnerability, it is acceptable since such vulnerabilities should be taken precedence over other vulnerabilities in practice.

Since different programming languages have different syntax, our work currently focuses on C/C++ language. However, our system could be extended to other programming languages by modifying syntax parsing related features according to the targeted programming languages. We leave the extensions to other languages as future work.

VII. CONCLUSION

This paper is the first one that looks into the characteristics of each type of security patches and proposes an automatic machine learning approach to classify security patches into vulnerability types. To collect adequate security patches for empirical analysis, we query the NVD and manually go through thousands of patches to filter out the noise. To achieve an in-depth analysis on the nature of security patches by type, we perform an empirical study on ten most common vulnerability types in C/C++ from three perspectives: patch localization, patch complexity, and patch flow changes. The analysis results confirm the importance of considering different types of security patches on vulnerability detection&mitigation. Further, we implement a machine learning-based approach that can be integrated into current vulnerability tracking systems to relieve software maintainers' burden on manual analysis and reduce the inconsistency caused by human subjectivity. The evaluation results show that our approach can achieve good performance.

References

[1] Synopsys, "2018 Open Source Software and Risk Analysis Report," https://www.synopsys.com/content/dam/synopsys/sigassets/reports/2018-ossra.pdf, 2018.

[2] White Source Software, "The state of open source vulnerabilities management," https://www.whitesourcesoftware.com/open-source-vulnerability-management-report/, 2019.

[3] S. Keßler and P. Alpar, "Customization of open source software in companies," in *IFIP International Conference on Open Source Systems*. Springer, 2009, pp. 129–142.

[4] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on firefox," in *Proceedings of the 8th working conference on mining software repositories*. ACM, 2011, pp. 93–102.

[5] 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*. ACM, 2017, pp. 2201–2215.

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

[7] Alexander Leonov, "CWEs in NVD CVE feed: analysis and complaints," https://avleonov.com/2017/10/21/cwes-in-nvd-cve-feed-analysis-and-complaints, 2017.

[8] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 869–885.

[9] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 481–490.

[10] MITRE, "2019 cwe top 25 most dangerous software errors," https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html, 2019.

[11] OWASP, "Owasp top 10 application security risks - 2017," https://www.owasp.org/index.php/Top_10-2017_Top_10, 2017.

[12] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 913–923.

[13] M. Soto, F. Thung, C.-P. Wong, C. Le Goues, and D. Lo, "A deeper look into bug fixes: patterns, replacements, deletions, and additions," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 512–515.

[14] W. E. Wong and V. Debroy, "A survey of software fault localization," *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45*, vol. 9, 2009.

[15] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries."

[16] S. Levin and A. Yehudai, "Boosting automatic commit classification into maintenance activities by utilizing source code changes," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 2017, pp. 97–106.

[17] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic classication of large changes into maintenance categories," in *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 2009, pp. 30–39.

[18] 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.

[19] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 201–213.

[20] 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 *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 426–437.

[21] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 887–902.

[22] D. Arya, W. Wang, J. L. Guo, and J. Cheng, "Analysis and detection of information types of open source software issue discussions," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 454–464.

[23] E. Aghaei and E. Al-shaer, "Threatzoom: neural network for automated vulnerability mitigation," in *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security*. ACM, 2019, p. 24.

[24] CVE Details, "Vulnerabilities By Type," https://www.cvedetails.com/vulnerabilities-by-types.php, 2019.

[25] X. Wang, K. Sun, A. Batcheller, and S. 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)*. IEEE, 2019, pp. 485–492.

[26] NIST, "NVD CWE Slice," https://nvd.nist.gov/vuln/categories, 2019.

[27] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 78–88.

[28] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[29] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.

[30] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2fix: Automatically generating bug fixes from bug reports," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 282–291.

[31] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2017.

[32] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *2012 34th international conference on software engineering (ICSE)*. IEEE, 2012, pp. 386–396.

[33] Z. Huang, M. DAngelo, D. Miyani, and D. Lie, "Talos: Neutralizing vulnerabilities with security workarounds for rapid response," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 618–635.

[34] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design of bug fixes," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 332–341.

[35] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[36] V. Pieterse and P. E. Black, *Algorithms and Theory of Computation Handbook*. CRC Press LLC, 1999.

[37] K. Lu, A. Pakki, and Q. Wu, "Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1769–1786.

[38] L. Kumar and A. Sureka, "Application of lssvm and smote on seven open source projects for predicting refactoring at class level," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, Dec 2017, pp. 90–99.