2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)

Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS

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

***Abstract*—Security patches in open source software (OSS) not only provide security fixes to identified vulnerabilities, but also make the vulnerable code public to the attackers. Therefore, armored attackers may misuse this information to launch N-day attacks on unpatched OSS versions. The best practice for preventing this type of N-day attacks is to keep upgrading the software to the latest version in no time. However, due to the concerns on reputation and easy software development management, software vendors may choose to secretly patch their vulnerabilities in a new version without reporting them to CVE or even providing any explicit description in their change logs. When those secretly patched vulnerabilities are being identified by armored attackers, they can be turned into powerful "0-day" attacks, which can be exploited to compromise not only unpatched version of the same software, but also similar types of OSS (e.g., SSL libraries) that may contain the same vulnerability due to code clone or similar design/implementation logic. Therefore, it is critical to identify secret security patches and downgrade the risk of those "0-day" attacks to at least "n-day" attacks. In this paper, we develop a defense system and implement a toolset to automatically identify secret security patches in open source software. To distinguish security patches from other patches, we first build a security patch database that contains more than 4700 security patches mapping to the records in CVE list. Next, we identify a set of features to help distinguish security patches from non-security ones using machine learning approaches. Finally, we use code clone identification mechanisms to discover similar patches or vulnerabilities in similar types of OSS. The experimental results show our approach can achieve good detection performance. A case study on OpenSSL, LibreSSL, and BoringSSL discovers 12 secret security patches.**

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

I. Introduction

Recent years have witnessed an impressive popularity of open source software (OSS). As one of the biggest hosting service providers, GitHub announced that there had been 31 million developers working across 96 million repositories in 2018 [6]. Meanwhile, the number of vulnerabilities in OSS continues to grow. A report from Snyk shows there is a 53.8% increase in the number of published open source vulnerabilities from 2016 to 2017 [25]. One reason is that the source code of OSS can be carefully analyzed by attackers to discover the unknown vulnerability. What's worse, the security patch of vulnerability exactly points out the vulnerable code, which teaches attackers how to generate exploits for attacking the unpatched version. For instance, just one day after the remote code execution vulnerability in Apache Struts 2 (CVE-2017-5638) was publicly disclosed and fixed, exploit scripts appeared in the wild. Later, due to its unpatched system, Equifax got attacked and millions of personal data including social security number were exposed [1].

Though timely patching the vulnerability is an effective defense against those "N-day" attack, there exist some challenges in real world. In many cases, security patches are included in a large software patch or new version with other types of patches, e.g., bug fixes and new features. Since applying software patch or updating to new version increases the service system downtime and introduces extra workload, admins or users tend to postpone updating their running software until a stable version is available or the security advisory like Common Vulnerabilities and Exposures (CVE) shows that there exists a severe security patch of vulnerability [13].

However, software vendors may secretly patch their vulnerabilities without creating CVE entries or even describing the security issue in its change log. One reason is the concern that too many CVE entries or vulnerability fixes in the change log may hurt the quality reputation of their software. In addition, they may intend to block the publication of related CVE entries until they think it is safe to publicly release them. However, since the related patch or new version has already been available, attackers can still carefully analyze the code changes from the patch directly or from the difference between two versions and then generate exploits to misuse these secretly-fixed security vulnerabilities. To defend this, developers and users need an approach to identify the existence of secret security patch in open source software so that they can update their software in time. Moreover, similar type of software may contain the same vulnerabilities since code clone is common in open source software and developers tend to make the same mistakes when solving difficult intellectual problems [11]. In this case, analyzing the security patch in one software (e.g., OpenSSL) can help identify and fix the corresponding vulnerabilities in other software (e.g., LibreSSL) with the similar functionality. We consider those vulnerabilities as one type of "0-day" vulnerabilities.

In this paper, we develop a machine-learning based mech-

This work is partially supported by the NSF grant CNS-1822094, IIP-1266147 and ONR grants N00014-16-1-3214, N00014-16-1-3216, and N00014-18-2893.

978-1-7281-0057-9/19/$31.00 ©2019 IEEE
DOI 10.1109/DSN.2019.00056

anism to help automatically identify secret security patches from the released software patches and the difference between two versions of the open source software. First, since there is no publicly available dataset on security patches, we create a new security patch dataset that contains 4702 security patches by crawling all the available reference links in CVE entries [4] from year 1999 to 2018. Since security patches may have different patterns when written in different programming languages, we focus on C/C++ languages and pick out 1636 security patches written in C/C++. Also, we randomly fetch 1636 non-security patches from GitHub repositories [6].

To identify security patches, we face one major challenge of identifying effective features to model the differences between security patches and non-security patches. Based on manual analysis of a majority of the collected security and non-security patches, we identify a set of 61 features that belong to three categories, namely, basic feature, syntactic feature, and semantic feature. Next, we develop a machine learning based approach to distinguish security patches from non-security patches using the set of identified features. To increase the detection accuracy, we adopt a voting algorithm that ensembles five popular classification algorithms including Random Forest, Bayes Net, Stochastic Gradient Descent (SGD), Sequential Minimal Optimization (SMO), and Bagging. We randomly choose 80% of our dataset as the training dataset and use the remaining 20% as the testing dataset. The experimental results show that our model can achieve a good performance with 79.6% true positive rate and 41.3% false positive rate.

To further evaluate the effectiveness of our system, we perform a case study on three open source SSL libraries, i.e., OpenSSL, LibreSSL, and BoringSSL, and discover 12 secret security patches, among which the longest latency between the secret patch and the public release is over two years.

II. System Overview

Figure 1 shows the overview of our system, which consists of three major steps. The first step is to construct a security patch dataset for (i) extracting useful features for the machine learning model, (ii) training a machine learning based security model in the training phase, and (iii) evaluating the effectiveness of the security model in the detection phase. By querying all CVE entries in 1999-2018, we crawl 4702 security patches from at least 898 open source projects. Among them, we focus on the 1636 security patches from projects written in C/C++. In addition, we randomly fetch 1636 non-security patches from GitHub repositories. Therefore, our dataset contains 1636 security patches and 1636 non-security patches.

The second step is to derive a set of basic, syntactic, and semantic features for the machine learning based security model. Some features are collected from previous related work, and other features are newly identified via manual observation of our security patch and non-security patch database.

In the third step, we adopt a voting algorithm that ensembles five popular classification algorithms including Random Forest, Bayes Net, Stochastic Gradient Descent(SGD), Sequential Minimal Optimization(SMO), and Bagging, to build the machine learning based model. We transform the features of each patch into a vector along with a label marking if the patch is security patch or non-security patch. To evaluate the system performance, we randomly choose 80% of our dataset as the training dataset and the remaining 20% as the testing dataset.

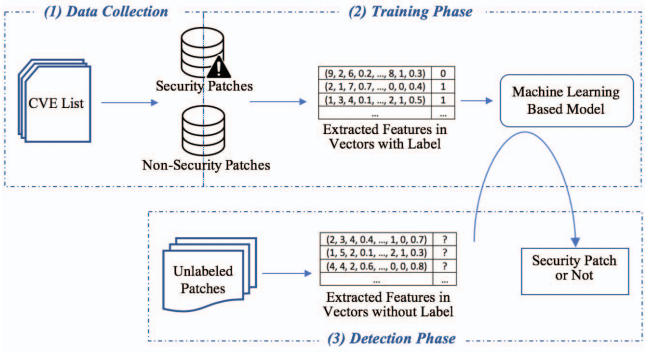

Fig. 1. System Overview

III. Patch Database Collection

Since there is no public available security patch database, we construct such as a database by querying CVE entries [4], whose reference links may contain the URL of the patches. Figure 2 illustrates an overall process of database collection.

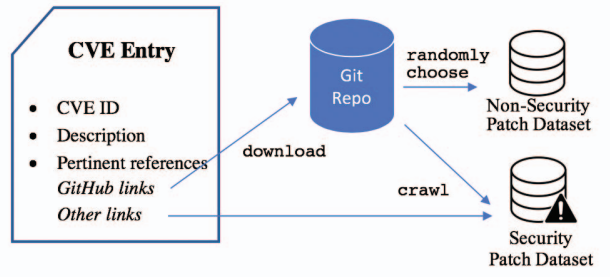

Fig. 2. Overview of Data Collection

To collect security patch dataset, we crawl all the related reference URLs that contain patches in all CVE entries from 1999 to 2018. To collect non-security patch dataset, we download open source repositories that appear in the CVE list and randomly choose the non-security commits as the non-security patch dataset [1]. We describe the methodology of data collection in the following.

A. Security Patch Dataset

Up to 04/11/2018, the CVE list consists of 126,491 CVE entries [4]. Each CVE entry includes a CVE ID, brief description of the vulnerability, and pertinent reference URLs of reports, advisories, and patches (if any). Based on our observation, the reference URLs can be divided into two categories according to the type of hosting

[1]The dataset is available at https://github.com/SecretPatch/Dataset

service providers, namely, projects hosted on GitHub and projects hosted on other websites. For the project hosted on GitHub, the reference URL of a security patch is in the form like: *https://github.com/owner/repo/commit/commit_hash*. The *commit_hash* is a hash value that is the unique identifier of a commit. The corresponding patch can be collected by downloading the commit URL with appendix *.patch*, i.e., *https://github.com/owner/repo/commit/commit_hash.patch*. On GitHub, one commit is corresponding to one patch. Therefore, each commit can be downloaded as a patch. We collect 4010 patches that belong to projects hosted on GitHub.

For the project hosted on other websites, e.g., its own website, there is no uniform form of reference URLs that can help tell if it contains any security patch. To solve this problem, we crawl all the reference URLs in CVE entries and use specific notations, i.e., *diff*, @@, +++, and −−− as indicators of existence of the patch. Once our crawler recognizes such notations, it removes the HTML/CSS labels and other unrelated contents in the web page (e.g., title, description, and etc.) and then downloads the remaining part as a security patch. Since many reference links before 2010 are no longer available, we only collect 692 security patches in this way.

Our model aims to identify the security patch through the syntactic and semantic patterns of the code. Since different programming languages have different patterns, we focus on patches of C/C++ projects that are very common in open source ecosystem. Our database contains 1636 security patches on projects written in C/C++.

```
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, filename)
6                /*
7                 * Store the resulting character for the 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, filename)
19           /*
20            * An ordinary character: store it in the 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);
30       }
```

Fig. 3. Patch Sample of CVE-2014-3158

B. Non-Security Patch Dataset

To train our model, we also need to collect a non-security patch dataset. Theoretically, a patch is corresponding to a vulnerability fix, bug fix, or feature update. However, due to different version control philosophies, some software vendors may release a big patch that mingles multiple patches. Also, for projects that do not host in control version system like GitHub, we may only generate a unified *.diff* file between an original and a modified source tree as a big patch that contains multiple patches. To avoid this problem, we collect the non-security patches from projects that appear in the CVE list and are hosted on GitHub, so that each commit is exactly one patch. After we download all 898 related GitHub repositories (about 1T size), for each project, we use the command `git log` to get all the commits with its hash value. To reduce the biases of specific projects, we randomly choose 1636 commits as the non-security patch dataset. Since the hash value is the unique identifier of a commit, we can filter out commits already included in the security patch dataset by comparing their commit hash values. Therefore, we obtain a security patch and non-security patch dataset of the same size.

IV. Security Patch Identification

From previous work and our observation, we collect a set of features that could distinguish between security patches and non-security patches in a machine learning model. Based on these features, each patch in our database can be represented as a vector with a label of security patch or non-security patch. After training a machine learning based model using the supervised dataset, when given a new unlabeled patch, our system can transform it into the corresponding vector and then identify if it is a security patch.

A. Feature Extraction

A patch [7] contains differences between old and new version files. Figure 3 shows an example of the patch for CVE-2014-3158. Each difference shown in a patch starts with a *diff a/folder_name/file_name b/folder_name/file_name* (e.g., line 1), and each difference may contain multiple change hunks that are continuous deleted and added lines marked with - and +, respectively. For instance, lines 9 and 10 is a change hunk.

Table 1 presents the features we collect in this work. We borrow Feature 1-22 from Tian et al.'s work [26], which shows that those features are effective on distinguishing vulnerability and bug fixing patches from new feature patches. They consider the changes in files, hunks, conditional statements, loops, lines, characters, and function calls. *Total* refers to the sum of removed and added number of these basic program features, and *Net* is the number of the added minus that of the deleted. In addition, our work aims to distinguish security patches from non-security patches, which requires more features to represent the difference between security patches (i.e., vulnerability fix) and non-security patches (i.e., bug fix and newly added feature). By manually comparing security and non-security patches in our database, we have the following observations:

- Security patches are more likely to modify less code than non-security patches.
- Security patches are more likely to introduce modifications on operators and operands. For instance, in

TABLE I
LIST OF FEATURES

No.	Description	Type
1	# of changed files	basic feature
2	# of hunks	
3 - 6	# of removed/added/total/net lines	
7 - 10	# of removed/added/total/net characters	
11 - 14	# of removed/added/total/net conditional statements	syntactic feature
15 - 18	# of removed/added/total/net loops	
19 - 22	# of removed/added/total/net function calls	
23 - 24	# of total/net modified functions	
25 - 28	# of removed/added/total/net arithmetic operators	
29 - 32	# of removed/added/total/net relation operators	
33 - 36	# of removed/added/total/net logical operators	
37 - 40	# of removed/added/total/net bitwise operators	
41 - 44	# of removed/added/total/net memory operators	
45 - 48	# of removed/added/total/net variables	
49 - 51	AVE/MIN/MAX Levenshtein distance within hunks (before abstraction)	
52 - 54	AVE/MIN/MAX Levenshtein distance within hunks (after abstraction)	
55	# of same hunk (before abstraction)	
56	# of same hunk (after abstraction)	
57 - 58	# and % of affected files	semantic feature
59 - 60	# and % of affected functions	
61	Any data dependency changes (True or False)	

vulnerability caused by boundary problem, change > into >= or change `n` to `n-1`.

- Security patches are more likely to move a piece of code to another place with no other changes. For instance, it is common to move a conditional statement inside a loop outside for security patch.
- In security patches, the same or similar change hunk may appear multiple times in different functions or files.

Therefore, we conclude 34 more syntactic features:

- **# of total/net modified functions.** Different from previous function calls which are represented by the function name or pointer in change hunks, the number of modified functions represents how many functions contain the change hunks. This number helps assess the direct affected range of a patch. For instance, for a patch which contains 3 change hunks within a function, this number is counted as 3 in total and 1 in net.
- **# of total/net/removed/added basic operators.** We count the total and net number of basic operators including arithmetic, relation, logical, and bitwise operators which occur in each patch. Also, we count these numbers in removed part and added part, respectively.
- **# of total/net/removed/added memory operators.** We count the corresponding number of C/C++ memory related operators which occur in each patch, e.g., `malloc`, `calloc`, `realloc`, `free`, `sizeof`, and etc.
- **AVE/MIN/MAX Levenshtein distance within hunks (before abstraction).** Levenshtein distance is a measure of the similarity [20]. In our work, Levenshtein distance within a change hunk is the number of deletions, insertions, and substitutions required to transform the removed hunk into added hunk. Since there are always many hunks within one patch, the average, minimum, and maximum Levenshtein distance among them are used to represent such characteristics of a patch.
- **AVE/MIN/MAX Levenshtein distance within hunks (after abstraction).** To further measure the similarity of each pair of removed and added hunks, we abstract the code. After removing the space and comment, we replace all the identifiers with `$`. Then, we calculate the corresponding Levenshtein distance on these abstracted code.
- **# of same hunks (before abstraction).** We consider every two exact same change hunks as a pair of same hunks.
- **# of same hunks (after abstraction).** To count the pair of similar hunks, we first remove the exact same hunk. Then, after abstracting the code, we regard every two same abstracted change hunks as a pair of similar hunks.

Moreover, we propose 5 semantic features:

- **# and % of affected files.** The number of affected files is computed by counting the number of files that call the modified functions in the given patch. The percentage is calculated by dividing the number of affected files with the total file number.
- **# and % of affected functions.** We compute the number and the range of affected functions from code property graph which combines control flow graph and data dependency graph generated by Joern [28]. We combine several nodes of a hunk as a node and then count the number of connected node in the function level to get the number of affected functions. The percentage is calculated by dividing the number of affected functions with the total function number.
- **Any changes of data dependency.** After combining several nodes within a hunk as a coarse-grained node, if any nodes connected or the variables in the connected edge change, this value is `true`. Otherwise, it is `false`.

TABLE II
PERFORMANCE ON COLLECTED DATASET

Training Dataset	Testing Dataset	TP (%)	FP (%)	Precision	Recall
2618	654	266 (79.6%)	132 (41.3%)	66.8%	79.6%

B. Machine Learning Model

We use a number of popular classification algorithms including Random Forest [8], Bayes Net [5], Stochastic Gradient Descent (SGD) [2], Sequential Minimal Optimization(SMO) [21], and Bagging [3], in our machine learning model. However, each individual classifier cannot perform well. To improve the performance, we adopt a voting algorithm that ensembles the above five classifiers to do the majority vote.

We randomly choose 80% security and non-security patch dataset and transform each of them into a vector of values on above 61 features with its label "1" (i.e., security patch) or "0" (i.e., non-security patch) as the input training data. In the detection phase, we transform the remaining 20% patches in our datasets into vectors and then apply our model. If a vector is assigned "1", the corresponding patch is detected as a security patch. Otherwise, the corresponding patch is detected as a non-security patch.

V. SYSTEM EVALUATION

To evaluate the effectiveness of our system, we conduct experiments in three ways. First, we split our database into training and testing dataset to evaluate the detection accuracy of our model. Second, we apply our model on patches of 20 minor versions of OpenSSL 1.0.1 and compare our results with previous work. Third, we extend our experiments to several popular SSL libraries (i.e., OpenSSL, LibreSSL, and BoringSSL) to discover a number of secret security patches.

A. Performance on Collected Dataset

We randomly choose 80% of our dataset as the training dataset and the remaining 20% as the testing dataset (334 positive samples and 320 negative samples). We adopt a 10-fold cross-validation to choose the best parameter configuration. Our experiments are conducted on a machine with 3.1 GHZ Intel Core i7 CPU and 16GB RAM. The training phase (including 2618 patches) takes 42s and the testing phase (including 654 patches) takes 9s. Table II shows the true positive (TP), false positive (FP), precision, and recall of our testing results, respectively. Our model can achieve 79.6% true positive rate and 41.3% false positive rate.

B. Performance on OpenSSL

We compare the effectiveness of our system with other security patch identification system. To the best of our knowledge, SPAIN [27] is the only work for this kind of research. Though SPAIN focuses on binary level patch analysis, it can be used to identify open source patches and it also conducts experiments on an open source project - OpenSSL to evaluate its accuracy. To compare with SPAIN, we apply our model on all the patches between 20 minor versions of OpenSSL 1.0.1 series (i.e., all the commits from OpenSSL 1.0.1 to 1.0.1s).

Table III presents the comparison between SPAIN and our work. The second and third columns show the number of security and non-security patches that SPAIN and we manually identify as the ground truth. The reason for the different identified numbers is that SPAIN can only identify the patches within one function with control flow changes, but our work extends it to inter-function patches. Actually, a patch may involve modifications across multiple functions that have impacts on each other. SPAIN may regard a patch of multiple function fixes as multiple patches whereas such patch is regarded as one patch in our work. In addition, our approach takes patches of header files into consideration while SPAIN does not.

The percentage of security patches we can identify is 8% higher than SPAIN. On the other side, our approach has a higher number of false positive (e.g., 190) than SPAIN (e.g., 47). We argue that our approach shows competitive results when comparing with previous work such as SPAIN. Our approach is able to cover inter-function patches, header file patches, and patches without control flow changes whereas SPAIN cannot. Besides, our approach has shown good performance and scalability on a larger number of vulnerabilities from various types of software in NVD, as shown in Table II. It is difficult for SPAIN to provide a similar performance result, due to the huge demanded efforts on obtaining the ground truth for various binary code.

C. Case Study: SSL Libraries

To identify secret security patches in wild, we extend our experiments to three open source SSL libraries, including OpenSSL, LibreSSL, and BoringSSL. First, we apply our toolset to identify the security patches from the commits of above three projects from GitHub. Once a security patch is identified, we use code clone algorithms [23], [24] to detect if this vulnerability has been patched in other projects.

Table IV summarizes 12 secret security patches and the fix information in these three SSL libraries. The first column shows the CVE ID of each security patch in one project. The Fix Date column of each project is obtained from the date in the patch (commit) of each vulnerability that represents the fix date. The grey background cell denotes the earliest fix date of each vulnerability. The dash means such vulnerability does not apply to this project. Each project's Lag Day is the date difference between the first fixed date of other projects and its fixed date, during which attackers can launch "0-day" attack on these similar type of software. "Not yet" means the project contains such vulnerability and it has not been fixed until now (12/06/2018). We get the second to the last column by manually checking the advisory in CVE entry or its official hosted website and then the Secret Day can be computed as date difference between the CVE ID belonging project's first fix date and the advisory release date, which can be utilized by attackers to attack unpatched versions.

TABLE III
COMPARISON WITH A PREVIOUS WORK ON OPENSSL ANALYSIS

Technique	Ground Truth		Detection Result			
	# Security Patch	# Non-Security Patch	# TP	# FP	Precision	Recall
SPAIN [27]	323	217	229	47	0.83	0.71
This work	294	365	231	190	0.55	0.79

TABLE IV
SECRET SECURITY PATCHES AMONG THREE SSL LIBRARIES

CVE ID	OpenSSL		LibreSSL		BoringSSL		Public Disclosure Date	Secret Day
	Fix Date	Lag Day	Fix Date	Lag Day	Fix Date	Lag Day		
CVE-2018-5407 (OpenSSL)	04/19/2018	-	Not yet	232+	-	-	11/15/2018	240
CVE-2018-0734 (OpenSSL)	10/23/2018	-	Not yet	45+	Not yet	45+	10/30/2018	7
CVE-2018-0732 (OpenSSL)	06/11/2018	974	06/13/2018	972	10/11/2015	-	06/12/2018	973
CVE-2018-0739 (OpenSSL)	03/22/2018	-	08/06/2018	137	03/27/2018	5	03/27/2018	5
CVE-2017-3731 (OpenSSL)	01/18/2017	-	02/01/2017	14	-	-	01/26/2017	8
CVE-2016-7053 (OpenSSL)	10/16/2016	849	07/11/2014	21	06/20/2014	-	11/10/2016	874
CVE-2016-7052 (OpenSSL)	08/22/2016	-	-	-	09/26/2016	35	09/26/2017	35
CVE-2016-6305 (OpenSSL)	09/10/2016	-	Not yet	818+	-	-	09/22/2016	12
CVE-2016-6304 (OpenSSL)	09/09/2016	-	09/27/2016	18	-	-	09/22/2016	13
CVE-2016-6308 (OpenSSL)	09/10/2016	-	Not yet	818+	-	-	09/22/2016	12
CVE-2018-8970 (LibreSSL)	01/22/2015	-	03/22/2018	1134	-	-	03/24/2018	2
CVE-2018-12434 (LibreSSL)	06/19/2018	982	06/13/2018	976	10/11/2015	-	06/14/2018	977

Below are several interesting phenomena we observe. First, though each vulnerability listed in the table IV existed in at least two projects, only one CVE ID was created for one of them, and all others just secretly patched those vulnerabilities.

Second, the *.../crypto/dsa/dsa_ossl.c* file in LibreSSL allows a memory-cache side-channel attack on ECDSA signatures. NVD published date of this vulnerability (CVE-2018-12434) was 06/14/2018 while the new version that contains the corresponding patch was released on LibreSSL's website one day early. Since only two changes were made in this new version, it is not hard for attackers to analyze and utilize it to attack unpatched version in one day. What's worse, OpenSSL contained the same vulnerability and it only released a patch on the GitHub one week later (06/19/2018) without reporting to CVE. And a new version that contains the patch was released 30 days later on its website.

When we tried to request a CVE ID, CVE website asked us to contact the corresponding participating CVE Numbering Authority (CNA), i.e., OpenSSL Software Foundation. However, they replied us that this vulnerability could only cause a local-host side channel attack, so no CVE was needed. In contrast, there is no participating CNAs for LibreSSL, anyone can request a CVE ID for LibreSSL by contacting MITRE Corporation directly. We can see that the CNA mechanism provides software vendors an opportunity to secretly patch their vulnerability without creating a CVE ID. In this case, when comparing OpenSSL with LibreSSL, users may draw a biased conclusion that OpenSSL is more secure since the number of its recent CVE is smaller.

Third, it may take long time for other projects to realize those secretly patched vulnerabilities and take actions. CVE-2018-0732, CVE-2016-7053, CVE-2016-6308, and CVE-2018-8970 show that the date difference between the first fix and a CVE entry assignment for another project is more than two years. Since these software vendors do not have a good channel to share the newly discovered vulnerabilities, attackers may misuse those "0-day" vulnerabilities for a long time.

Lastly, the software version control process should announce security fixes more clearly and accurately. In CVE-2018-8970, the *int_x509_param_set_hosts* function in *x509_vpm.c* file does not support a certain special case of a zero name length, which causes silent omission of host name verification. This can be exploited to launch man-in-the-middle attack to spoof servers and obtain sensitive information via a crafted certificate. Though there was a CVE ID assigned to LibreSSL for this issue, LibreSSL described this as a bug fix in its change log without mentioning any security related issues. However, LibreSSL usually explicitly classifies all the patches into security fix, bug fix, and new feature in its change log. In this case, when a user only focuses on its change log, it may bypass one patch since there is no vulnerability fix and other small bugs are tolerable in the system.

VI. DISCUSSION AND LIMITATIONS

There may exist a long distance from identifying a suspicious vulnerability to truly triggering it as a real attack. It has been reported that not all the vulnerabilities reported in CVE have known approaches to trigger them [18]. In addition, since a number of vulnerabilities without CVE IDs have been triggered, it indicates the not all the security patches have corresponding CVE IDs. We collect our security patch database from all the available patch source in CVE list, which may be biased towards some types of severe vulnerabilities. For instance, we mention that OpenSSL refuses to assign a

CVE ID for a local-host side channel vulnerability due to the low exploitability in their opinion. Since the manual triggering of suspicious vulnerabilities and evaluating their severity may demand huge efforts and domain experts, we consider how to identify exploitable vulnerabilities as a future work.

It is possible that our approach may be leveraged by attackers. Actually, we wonder attackers might have already misused secret patches to some extend. The goal of our work is to promote software vendors to maintain their products more normatively, increase the collaboration between software vendors on information sharing, and finally eliminate this type of "0-day" attacks.

The non-security patch dataset may still contain some security patches due to the unknown secret security patches, and it may introduce impacts to our experimental results. We manually check 536 out of 1636 patches randomly chosen from non-security patch dataset and identify 7% of them as security fixes based on our domain knowledge. Similar to previous work [14] [19], we argue this rate is acceptable since machine learning is capable of dealing with noisy datasets. In the future, we will further clean the dataset by removing those security patches from the non-security patch dataset.

The present design of our system uses commits on GitHub as the smallest unit of the patch. In practice, although GitHub is one of the most popular open source software hosting service provider, not all of the open source software is hosted on GitHub. For open source software hosted on other websites, patches can only be acquired from a *diff* file of the source code of neighboring versions. However, the *diff* file may consist of a number of change hunks belongs to multiple patches. For a small *diff* file, we can simply separate it through keyword matching. However, when this file is large, for instance, a main version introduces many modifications, it is hard to separate hunks into individual patches. We leave it as our future work.

Currently, our system only supports to identify security patch in open source projects written in C/C++. Our system can be adapted to other programming languages by modifying features, e.g., syntax parsing related features 11-56. In our future work, we plan to extend it to OSS projects written in other types of programming languages and even multiple programming languages.

VII. Related Work

OSS vulnerability detection has become an active research area. There are two main research directions: vulnerable code similarity comparison and vulnerability pattern recognition. For vulnerable code similarity detection, the traditional token-based techniques remove all the whitespace and comments and replace variable and function names with a specific character to detect Type-1 and Type-2 code clone [22] that only makes few modifications of identifiers, comments, and whitespace. The tree-based techniques [9], [29] mainly transform the program into Abstract Syntax Tree (AST) and then compare the longest common sequence (LST). Graph-based techniques [12], [17] use control and data dependence graph to detect code clones as isomorphic subgraphs. For vulnerability pattern recognition, machine learning or deep learning approaches are proposed by extracting the patterns from the vulnerable code and then searching the code with the same pattern. VulPecker [15] uses different sets of features to detect different types of software vulnerabilities. VulDeepecker [16] trained a neutral network to detect buffer overflow and resource management errors caused by library/API call.

Some work has attempted to create a database of security patches. Seulbae et al. [10] collected data from eight well-known Git repositories, and Zhen et al. [15] built a Vulnerability Patch Database (VPD) from 19 products. However, the size of these datasets are not sufficient to perform a machine learning based study, when comparing to the thousands of CVE entries on open source projects. Though Li et al. [14] built a large-scale security patch database based on the Git related records in NVD [4], they have not made their database available to the public yet.

Certain secret security patches have been reported ad hoc in different studies. Zhen et al. [16] found Xen silently patches the vulnerability after the disclosure of CVE-2016-9104 in Qemu. Their results also revealed the secret security patches between Seamonkey and Firefox (CVE-2015-4517) as well as between Libav and FFmeng (CVE-2014-2263). It motivates us to perform a study on the secret security patches.

Some researchers have paid attention to patch analysis. Zame et al. [30] made a case study on the difference between security and performance patches in Mozilla Firefox. Perl et al. [19] showed many statistic difference between vulnerability contributing commits and other commits. However, they cannot distinguish vulnerability fixes from non-security bug fixes. Frank et al. [14] conducted the first large-scale empirical study between security patches and non-security bug fixes, and it provides analysis on the basic characteristics of security patch. Xu et al. [27] presented a scalable approach to identify security patches through the semantic analysis of execution traces. However, it cannot handle cross-function security patches and does not perform well on identifying non-security patches that are similar to security patches.

VIII. Conclusion

In this paper, we develop a machine learning based security patch identification system. Developers and users can use our system to help identify secret security patches and decide if it is the time to update to the new version or apply the patches. We point out that once a security patch is identified, its corresponding vulnerability should be detected in other similar types of software and if identified, this patch can be utilized to patch similar vulnerabilities. To evaluate the effectiveness of our model, we build a database that is composed of the security patches in the CVE list. We make it open-source to promote public research on improving the security of OSS. We discover a set of syntactic and semantic code features to profile security patches. The experimental results show that our system can achieve a good detection performance. We also apply our system to three open source SSL library projects and discover 12 secret security patches.

REFERENCES

[1] "Equifax offically has no excuse," https://https://www.wired.com/story/equifax-breach-no-excuse/.

[2] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[3] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[4] Common Vulnerabilities and Exposures (CVE), https://cve.mitre.org/cve/identifiers/index.html.

[5] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian network classifiers," *Machine learning*, vol. 29, no. 2-3, pp. 131–163, 1997.

[6] Github, "The state of the octoverse 2018," https://octoverse.github.com.

[7] GNU Diffutils, https://www.gnu.org/software/diffutils/.

[8] T. K. Ho, "Random decision forests," in *Document analysis and recognition, 1995., proceedings of the third international conference on*, vol. 1. IEEE, 1995, pp. 278–282.

[9] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

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

[11] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Transactions on software engineering*, no. 1, pp. 96–109, 1986.

[12] J. Krinke, "Identifying similar code with program dependence graphs," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 2001, pp. 301–309.

[13] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[14] 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.

[15] 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.

[16] 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.

[17] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 872–881.

[18] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*. USENIX, 2018, pp. 919–936.

[19] 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.

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

[21] J. C. Platt, "12 fast training of support vector machines using sequential minimal optimization," *Advances in kernel methods*, pp. 185–208, 1999.

[22] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queens School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[23] SIM, https://dickgrune.com/Programs/similarity_tester/.

[24] Simian, https://www.harukizaemon.com/simian/.

[25] Snyk, "The state of open source security 2018," https://snyk.io/stateofossecurity/.

[26] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 386–396.

[27] 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.

[28] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 590–604.

[29] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991.

[30] 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.