VCIPR: Vulnerable Code is Identifiable When a Patch is Released (Hacker's Perspective)

Junaid Akram*(Member, IEEE), Qi Liang*, and Luo Ping*

*State Key Laboratory of Information Security, School of Software Engineering, Tsinghua University, Beijing China.

Email: [znd15,liangq15]@mails.tsinghua.edu.cn

Email: luop@mail.tsinghua.edu.cn

Abstract—**Vulnerable source code fragments remain unfixed for many years and they always propagate to other systems. Unfortunately, this happens often, when patch files are not propagated to all vulnerable code clones. An unpatched bug is a critical security problem, which should be detected and repaired as early as possible. In this paper, we present VCIPR, a scalable system for vulnerability detection in unpatched source code. We present a unique way, that uses a fast, token-based approach to detect vulnerabilities at function level granularity. This approach is language independent, which supports multiple programming languages including Java, C/C++, JavaScript. VCIPR detects most common repair patterns in patch files for the vulnerability code evaluation. We build fingerprint index of top critical CVE's source code, which were retrieved from a reliable source. Then we detect unpatched (vulnerable/non-vulnerable) code fragments in common open source software with high accuracy. A comparison with the state-of-the-art tools proves the effectiveness, efficiency and scalability of our approach. Furthermore, this paper shows that how the hackers can easily identify the vulnerable software whenever a patch file is released.**

Keywords: Vulnerability detection, Maintaining code, Patch files, Plagiarism detection, Unpatched code clone, Vulnerable code fragments

I. INTRODUCTION

The security of software is under considerable scrutiny in recent years. Vulnerability detection and patching study has improved the ability to quickly discover and repair vulnerable code patterns. However, most of the research work on security measures has been qualitative, which focused on the prevention of vulnerabilities in these software systems. A bug, CVE-2017-5638, which was revealed in March 2017, but the hackers were using it till late May 2017 against Equifax[1] and they were stealing personal information including names, birth dates and social security numbers, etc. of almost 143 million users in the US [1]. Another bug CVE-2018-1274 released in 2018-04-10 of critical severity, which cause a denial of services (DOS) attack to allow unlimited resource allocation to an authenticated remote user (attacker) for parsing, which actually causes a DOS attack on CPU and memory consumption. It was fixed and a patch was released on 2018-04-17[2]. But a lot of people are still unaware of this vulnerability, even some peoples just ignore the patch files without knowing its importance. In another case there are many developers, who have used the same code snipped from other software systems

without applying vulnerability patches to it and delivered the product to the customers. These customers are just normal computer users and they are unaware of the term vulnerability, so they use their system at risk all the time. An automatic exploit is possible through path files [2]. Patches to vulnerable code are very important to make sure security measures [3] [4] [5]. Previous work shows that when a patch file is released, an attacker can easily use this patch to find the buggy code and in a few seconds automatically create an exploit [6].

Open source software (OSS) has increased the rate of vulnerabilities because as the name implies that the code is open source and available to everyone. Most of the software developers copy the code from other software systems and reuse them without modification, this type of reuse code is called *cloned code*. It is very adapting approach during development [7]. However, during or after development process it is quite difficult to say which code fragment is the original and which was copied. Clones in source code actually cause issues in security maintenance [8]. Previous research work shows that almost 7% - 23% of source code is actually cloned in large systems [9]. Even recent research work [10] [11] on very large systems [12] have shown that 22.3% of Linux code has been cloned. This cloned code can be a vulnerable code, which was repaired by patch file in the original source file but still unpatched in other systems [6]. In particular, OSS are widely used code-bases in development, So, code cloning is actually becoming a major causes of vulnerabilities. For example, OpenSSL a Heart-bleed vulnerability (CVE-2014-0160) affected many types of systems, i.e websites, web applications & servers, operating systems and applications, because most of the affected system at that time either used the whole library of OpenSSL or copied some part of it in their systems [13].

In this paper, we present VCIPR, a lightweight and scalable vulnerability detection technique for unpatched (vulnerable) code clone files. The vulnerable code was retrieved from the patch files and their original source file, which was used as a signature set to detect similar vulnerable code fragments in other open source systems. It has been explored that, vulnerable code can be identifiable and extractable when a patch is released. So, it's very important to find unpatched code clones in other software as early as possible. For the experimental purposes of our approach, we collected more than 1,000 CVE based already published vulnerabilities from different open

---

[1]https://www.engadget.com/2017/09/13/equifax-apache-argentina/
[2]https://pivotal.io/security/cve-2018-1274

source software, including Linux kernel, OpenSSL, Apache Server etc.

We detect most common repair patterns in patch files, which helps us know more about why changes required through patch files to make sure security measures. We adopt a novel mechanism to index fingerprint of vulnerable code fragments at function level granularity. VCIPR took almost 12 hours to process a billion LOC, which is quite fast as compared to previous techniques. It is an efficient technique for detecting unknown vulnerable clones with high accuracy of almost 83%. Furthermore, the accuracy of our technique also depends on the exact collection of know vulnerabilities and the vulnerable function extraction technique adopted by us. This paper answers many short questions, which strike into our brain when we talk about patch files, i.e. evaluation the stability of patch files, privacy issue in patch files, patch files analysis, feature extraction from vulnerable code, why did change come in patch file?, similar vulnerabilities and reason to change the code, etc. In this paper, our contributions are as follows:

- Vulnerability source code collection against it's CVE reference number.
- Detection of most common repair patterns in patch files.
- Preprocessing, feature extraction and fingerprint generation of vulnerable code.
- Unpatched (vulnerable/ non-vulnerable) code clones detection.
- Security-related bugs evaluation study.
- Conduct experiments to evaluate our approach.
- Comparison with the other state-of-the-art-tools.

Here are some research questions discussed in this paper:

RQ1: Are all the repair patterns in patch files designed to remove the vulnerability issue?

RQ2: Is it possible to detect the unpatched (vulnerable) code in open source systems?

RQ3: Do the unpatched (non-vulnerable) code clones affect the security measures?

RQ4: Is scalability an issue in vulnerability detection?

A. Related Terms and Definitions

Here are some terms, which are used in this paper.

Vulnerability: A weakness or a flaw in a software system, which allows an attacker to overcome a system's security. It is the intersection of system susceptibility and capability to exploit this flaw.

CVE: It is the Common Vulnerabilities and Exposures[3] system, which provides a reference by assigning CVE-IDs for publicly known security vulnerabilities. Each containing a unified identification number, description, and minimum one public reference for known cybersecurity vulnerabilities.

NVD: The NVD[4] (National Vulnerability Database) is the United States government standards based vulnerability management repository. This enables the automation of security

Fig. 1: Chunk after preprocessing

measurements, vulnerability management and compliance. It includes security references, security flaws, misconfiguration, product names and impact metrics.

Code Clones: A copied code fragment from other software systems is called code clone. There are four different types of code clones [12], i.e type-1: Exact clones, type-2: Renamed clones, type-3: Restructured clones and type-4: Semantic clones.

Token: This is the minimum preprocessed unit that compiler can understand. For example, in *(int x = 10 ;)* there are five tokens in it, which we converted into *(id0 id1 = 0;)* during preprocessing and normalization.

Chunk: It is a set of tokens, as shown in right side of Fig 1.

Indexing: Indexing information consists of fingerprint hash values of vulnerable code saved in HBase.

II. VCIPR METHODOLOGY

In this section, we explain the main idea behind VCIPR vulnerability detection technique, which is a scalable approach that can be implemented at a large scale level on massive open source software. We proposed different stages to detect unpatched code clones at a large scale, these stages have been explained below and the top level architecture has been shown in Fig 2. The choices of VCIPR vulnerability detection technique are motivated by these goals:

1) Focusing on detection of vulnerable code fragments
2) Scaling to large code bases
3) Minimizing the false positive rate
4) Support multiple languages

The VCIPR accomplish these goals by using the following steps:

A. CVE Collection and Vulnerability Code

For the experiment purposes, we collected more than 1000 top CVE security exposures. For an overview of CVE patch file, an example of CVE-2018-1066 patch file has been shown in Fig 3. CVE-2018-1066[5] was released in February 2018. This vulnerability allows a hacker to control a CIFS Server. In Fig 3, It has been mentioned that there are two files have been changed to overcome this vulnerability. A vulnerability has been patched through a patch file, and a patch file may have contained multiple diff files. However, in the same diff
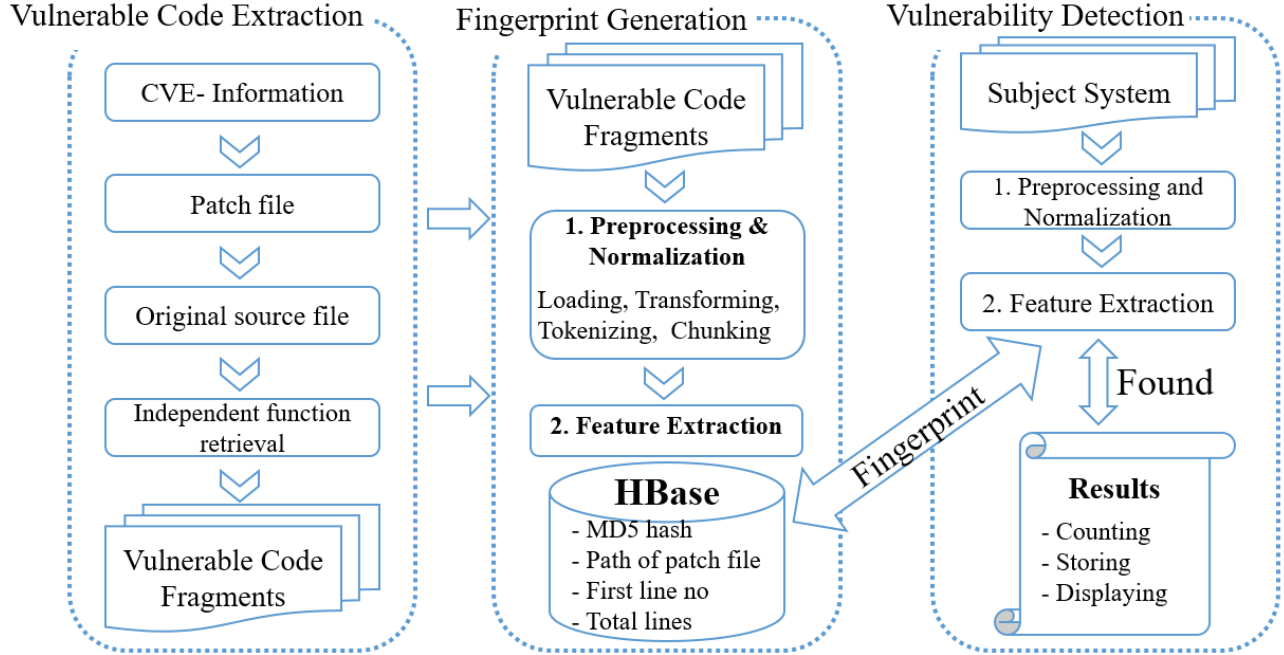
Fig. 2: Main architecture (work-flow) of VCIPR

file, multiple code modifications may have been formed. The number of unmodified rows and the code modification section is called a hunk. Therefore, the diff file in Fig 3 shows one hunk. The addition of code is displayed in green color using + symbol and the deletion of code in these files displayed in red color using − symbol. So, it's quite easy for an attacker to get the vulnerable code and use it for criminal purposes. But it's still not an easy process because these shown code fragments are not independent functions. They are maybe a part of function, which has been called somewhere else in an other class. Some vulnerability codes exist only in one function of one file, some of them scattered in different functions, some of them even exist across different files. The patch file includes the records to modify the code to fix the vulnerability. So, to solve this problem and retrieve the vulnerable code, we open a new chapter by exploring their original source code files, where this patch file should be applied and committed. From those original source files, we retrieve almost 3,935 independent functions and code fragments which are in fact vulnerable and can be used as a fingerprint.

B. Detection of most common repair patterns

By exploring and studying the commit changes of almost 2000 CVE patch files of Linux kernel [6], which are the part of our vulnerable source code data-set. We detected some more common repair patterns happened in the code, as shown in Fig 4.

By locating the vulnerable source code files from patch files, we extract the vulnerable functions as it has already

[6]https://git.kernel.org/

Fig. 3: One hunk of a CVE-2018-1066 patch file

been explained in previous section. So during the extraction process, we evaluated these source code changes in patch files. These changes appears in the form of most common repair patterns, i.e changes in condition blocks, addition of new lines, deletion of lines, changes in reference values, fixing expression, removing null values, etc. So, we counted these patterns from almost 2000 patch files. Fig 4 displays all of these patterns, where + symbol shows the addition in code and the − symbol shows the deletion in code. These most common repair patterns are explained below.

Addition of Conditional_block: The majority of bugs in source code require more addition than changing or removing the code. Conditional blocks with *return* statement also shows

Addition of Conditional_block

```
8563 @@ -1371,6 +1371,9 @@ int kvm_vgic_inject_irq(struct kvm *kvm, int cpuid, unsigned i
8564          goto out;
8565     }
8566 +   if (irq_num >= kvm->arch.vgic.nr_irqs)
8567 +       return -EINVAL;
8568 +
8569     vcpu_id = vgic_update_irq_pending(kvm, cpuid, irq_num, level);
8570     if (vcpu_id >= 0) {
8571         /* kick the specified vcpu */
```

Expression_fixing

```
111 @@ -1027,7 +1027,7 @@ bool dev_valid_name(const char *name)
112 {
113     if (*name == '\0')
114         return false;
115 -   if (strlen(name) >= IFNAMSIZ)
116 +   if (strnlen(name, IFNAMSIZ) == IFNAMSIZ)
117         return false;
118     if (!strcmp(name, ".") || !strcmp(name, ".."))
119         return false;
```

Return_value and Reference_change

```
4917 @@ -216,9 +217,11 @@ static int twl4030_madc_battery_probe(struct platform_device *pdev)
4918     twl4030_madc_bat->pdata = pdata;
4919     platform_set_drvdata(pdev, twl4030_madc_bat);
4920 -   power_supply_register(&pdev->dev, &twl4030_madc_bat->psy);
4921 +   ret = power_supply_register(&pdev->dev, &twl4030_madc_bat->psy);
4922 +   if (ret < 0)
4923 +       kfree(twl4030_madc_bat);
4924 -   return 0;
4925 +   return ret;
```

Change in Constants

```
8392 @@ -168,7 +168,7 @@ static const struct snd_pcm_hw_constraint_list
8393 };
8394 static const unsigned int rates_22579[] = {
8395 -    44100, 88235, 1764000
8396 +    44100, 88200, 176400
8397 };
```

Removing_null

```
2685 @@ -461,8 +461,8 @@ static int s3c24xx_iis_dev_probe(struct platform_device
2686         return -ENOENT;
2687     }
2688     s3c24xx_i2s.regs = devm_ioremap_resource(&pdev->dev, res);
2689 -   if (s3c24xx_i2s.regs == NULL)
2690 -       return -ENXIO;
2691 +   if (IS_ERR(s3c24xx_i2s.regs))
2692 +       return PTR_ERR(s3c24xx_i2s.regs);
2693
2694     s3c24xx_i2s_pcm_stereo_out.dma_addr = res->start + S3C2410_IISFIFO;
2695     s3c24xx_i2s_pcm_stereo_in.dma_addr = res->start + S3C2410_IISFIFO;
```

Single_line_addition

```
6232 @@ -166,7 +166,7 @@ static void uhci_check_ports(struct uhci_hcd *uhci)
6233     /* Port received a wakeup request */
6234     set_bit(port, &uhci->resuming_ports);
6235     uhci->ports_timeout = jiffies +
6236 -   msecs_to_jiffies(25);
6237 +   msecs_to_jiffies(USB_RESUME_TIMEOUT);
6238     usb_hcd_start_port_resume(
6239     &uhci_to_hcd(uhci)->self, port);
```

Unwrapping the code

```
5072 @@ -245,7 +245,18 @@ static int spidev_message(struct spidev_data *spidev,
5073     k_tmp->len = u_tmp->len;
5074     total += k_tmp->len;
5075 -   if (total > bufsiz) {
5076 +   /* Check total length of transfers.  Also check each
5077 +    * transfer length to avoid arithmetic overflow.
5078 +    */
5079 +   if (total > bufsiz || k_tmp->len > bufsiz) {
5080         status = -EMSGSIZE;
5081         goto done;
5082     }
```

Fig. 4: Most common repair patterns

Fig. 5: The repair patterns ratio in 2000 patches

in the 4th part of Fig 4.

Removing_null: This repair is related to removing or adding null-checks. The 5th part of Fig 4 shows a negative null-check, which was removed through patch file.

Single_line_addition: It has been seen that in many patch files only a single line of code was added or removed to solve the security problem. An example has been shown in the 6th part of Fig 4.

Unwrapping the code: This pattern is similar to the conditional block pattern, the difference is to do wrapping or unwrapping of existing code with the help of conditional branch as shown in the last part of Fig 4. Sometimes, the wrapping goes beyond conditional statements by adding try-catch block. Adding a loop is also the part of wrapping process to turn the code from simple sequence to repeating ones.

Fig 5 presents the repair patterns ranking (Pie-chart) as they occur in almost 2000 patch files. The *Conditional_block* is the most prevalent pattern found, which was repaired in the patches, followed by *Expression_fixing* and wrapping/unwrapping the code. On the other hand, *Single_line_addition*, *Change in Constants* and *Removing_null* are less prevalent among the patch repair patterns.

C. Preprocessing and Normalization

This is a first but very important part of vulnerability detection in unpatched code. Vulnerability source code was derived from the unpatched source code files. During preprocessing, we split the source code into a sequence of tokens. Meanwhile, we replace all integers, variables, functions, methods etc, into some specified ids and numbers as shown in Fig 1. Preprocessing of vulnerable code includes many steps, i.e. loading, verifying, cleaning and tokenizing. The main contents of specific standardized preprocessing are as follows:

1) Removal of space, annotated content, etc
2) Removal of commands, this calls, namespaces, package reference symbols, etc.
3) Replace the identifier with id0, Id1, Id2, etc. and each code statement is calculated from 0
4) Replace string content with an empty string
5) Replace character content with fixed characters
6) Replace integer values with numeric 0
7) Replace all floating-point values with 0
8) Replace all boolean types with true

the addition pattern in the code. It involves the addition of new *if-then* block in the code as shown in Fig 4.

Expression_fixing: This pattern occurs to fix the existing logic or arithmetic expressions. The expression fixing is shown in the second part of Fig 4. The modification occurs when an existing expression is not functional or logic is preserved.

Return_value and Reference_change: Sometimes a reference was defined wrong and a method or a variable call is referenced by mistake. During patching, the wrong reference is replaced by the correct one. Return value referenced wrong is also the part of this as shown in the 3rd part of Fig 4.

Change in Constants: Change in constant values have also been seen more common in patch files, an example is shown
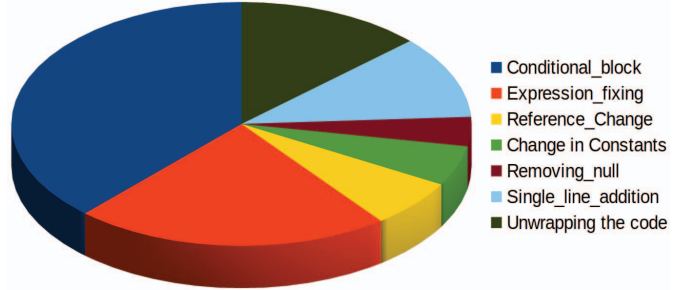
Fig. 6: Fingerprint information in HBase

D. Feature Extraction

A core phase of vulnerability detection was feature extraction from vulnerable source code to build fingerprints. These features include MD5 hash value, path of the patch file, first line number of chunk and total lines of the vulnerable code fragment. The chunk formation was performed on every file by putting tokens of a function into a single chunk. The MD5 hashing algorithm applied to get the hash value of the concerned vulnerable code fragment. The repeated hashes ignored by keeping it's patch file name, starting line and ending line number. Each feature of vulnerability code resulted as a fingerprint. Here are the main extracted features.

MD5 Hash: This is the hashing value of each chunk.

File Path: Location where the patch file is w.r.t its CVE-number

First Line Number: First line number of every chunk, to keep a record that from where this chunk starts. It helps us to retrieve vulnerable fragments.

Total Lines: To verify that the vulnerable code function has the same LOC or not.

E. Fingerprint Generation of Vulnerable Code

The fingerprint records the characteristics of the vulnerable code including the position of the code in the file, the language of the code, the path of the file, and so on. By using this method, the features of vulnerable code extracted for lexical analysis, rather than the semantic and grammatical analysis. So, a continuous segment of token stream formed a chunk, whose size is defined as *ck*, which means that a chunk contains *s* tokens. For the vulnerability code dataset, two adjacent chunks do not contain the same token, that is, the last token in the previous chunk is adjacent to the first token in the next chunk. The pseudo-code generated by vulnerability code fingerprints is shown in Algorithm 1. Input is the token flow formed after preprocessing and the output is chunk sequence. The analysis shows that the time complexity of the algorithm is $O(n)$, where *n* is the number of tokens in the token flow.

We generate the fingerprints of all the extracted vulnerable functions from every individual file by tracing its patch information. We consider every individual function as a single chunk in our data-set and generate its MD5 hash value. Fig 6, shows the fingerprint information saved in HBase, which consists on the Hash value (underlined), CVE-number, hunk number, file path, file name, first line number and total lines, which are encircled in red color.

Algorithm 1 Fingerprint generation algorithm of code

Input: File path, Function code.
Output: Generate chunks and extract main features, i.e. MD5 hash values, list of chunks, first & last line number of every chunk and size. Then add all these features in HBase.

1: **for** $i \Leftarrow 0\ to\ units.size() - 1$
2:   **if** $(units.get(i).hashCode())$ **then**
3:     $rowNum + +$
4:     $rowRecords.add(i)$
5:   **end if**
6: **end for**
7: **if** $(rowActualNum \geq leastRow)$ **then**
8:   $chunkList \Leftarrow \emptyset$
9:   $digest \Leftarrow buildHash(units, 0, row.get(size - 1))$
10:   $chunk \Leftarrow$ **new** $Chunk(element.getPath(), rowNum)$
11:   $chunkList.add(chunk)$
12:   **for** $i \Leftarrow 0\ to\ row.size() - sizePerChunk - 1$
13:     $digest \Leftarrow buildHash(units, row.get(i))$
14:     $chunk \Leftarrow newChunk(element.getPath(), digest, i + 1, rowNum)$
15:     $chunkList.add(chunk)$
16:   **end for**
17: **end if**
18: $/ * insert\ the\ featuresList\ into\ HBase * /$
19: $insertChunks(featuresList)$

Subject System | **Chunking & Hashing**

File 3
File 2
File 1

Function 1
Function 2
Function 3
⋮
Function n

Chunk 1
```
{
id0 id1 = id2 == null || id3 == 0;
id0 id1, id2;
id0 id1, 0, 0; id0 id1, 0, 0;
id0 = id1;
id0 = id1;
id0 = id1 id2 : id3;
if id0 {
     .....
```
MD5: 6fc97cb51737fd243b41e8d7735def4d

Chunk n
```
{
id0 id1 = id2 == null || id3 == 0;
id0 id1, id2;
id0 id1, 0, 0; id0 id1, 0, 0;
id0 = id1;
if id0 {
     .....
```
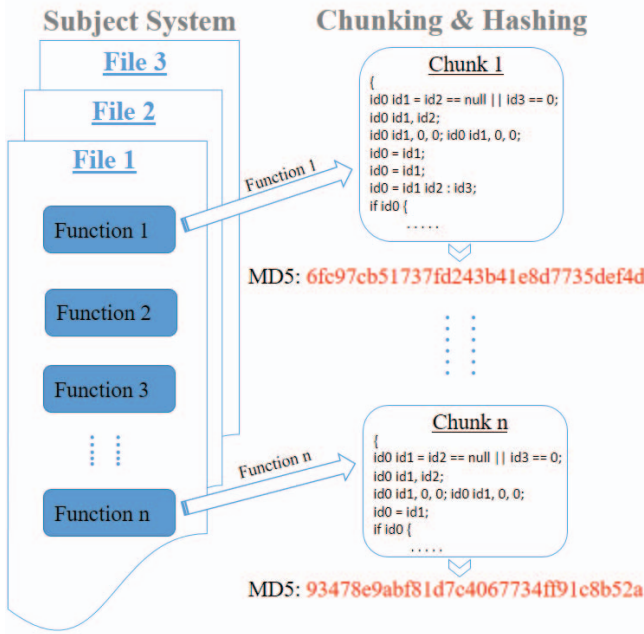MD5: 93478e9abf81d7c4067734ff91c8b52a

Fig. 7: Example chunk generation from subject system

F. Vulnerability Detection and Retrieval

In this section, the core detection algorithm describes the vulnerability detection and its clone relationship between the subject system and the fingerprint values. Finding all vulnerable code fragments was tricky and it involved a careful consideration. For example, how many tokens in one chunk need to be similar to consider it as a vulnerability? Is one line in unpatched clone file is enough, or we should consider multiple lines matching in one file? Does whitespace or comments matter? Do the sequence and order of statements matter, if it does, should we just consider some syntactic functions? Should we consider the text, tokens, or the parse? What if the two segments are equivalent in naming variables? etc. All of these questions have been involved in developing the vulnerability detection while assuring high accuracy and efficiency.

At the initial stage of the vulnerability detection process, we extract all the independent functions from every source code file of the subject system (testing project). Then the preprocessing and normalization has been performed on every extracted source code fragment of the subject system. After that, we formed the chunks in similar ways as it describes in the previous section of *Fingerprint Generation*. But during the detection process, the chunk formation of the subject system is a little bit different than fingerprint generation as shown in Fig 7. Because one subject system (testing project) can consists on multiple files $F1, F2, F3, ..., Fn$ and each file can consist on multiple functions. We perform preprocessing and normalization on every function and create a list of tokens and then put these sequence of tokens into a chunk. So, we consider every function as a chunk $C1, C2, ..., Cn$ and

generate its MD5 hash value as shown in Fig 7. Finally, we perform the scanning and hash matching in our vulnerability data-set fingerprint. The scanner scanned and retrieve all the hash values which were similar in HBase. Later on, we trace and locate these similar code fragments in subject system, which are detected as vulnerable code fragments. The vulnerable code was detected by two way.

1) The first way by building fingerprint index of all collected CVE's vulnerable source code, and then use an open source system as a subject system to detect similar code fragments in it.
2) The second way by building fingerprint index of many open source software, and then use the CVE's vulnerable code as a subject system to detect similar code fragments in an open source software.

So, through our technique, we detected vulnerable code in both ways. The scalability is not an issue in our technique because we used Hadoop file system, MapReduce and HBase to overcome the scalability and speed issue. In case of detection vulnerable code fragments in large scale-system, the MapReduce mainly divides the big task into several small tasks and then assigns these tasks to each map node. A piece of the source file of a subject system corresponds as a task. To overcome the communication overhead, the MapReduce allocate data distribution first to the local node. During the detection process, the map pretreatment, chunk calculation and feature extraction performed on every subject system by itself. The scanning and hash matching process retrieves all the similar hashes from HBase. Results according to similar hash values are written in the intermediate file. Then the reduce reads this file, which is produced by the map to display all the vulnerable code fragments.

G. Locating Vulnerable Code Fragments

This section addresses, the exact location of vulnerable code in the subject system and in the indexing table (HBase) with respect to it's CVE reference number. After the calculation of vulnerable code fragments by the MapReduce framework, we still need to locate the specific vulnerable code fragment. The basic method used is to mark out all the vulnerable chunks of subject system which are similar in index fingerprint repository, then merge all of them to display. If the files of subject system have n chunks, the corresponds chunks to the m in vulnerability source repository, then the time complexity is O (nm). If the vulnerable source repository is very big, the speed of detection process fall slow. So, to improve the vulnerability detection speed, we apply the chunks merging algorithm, which helps to get the linear time complexity. The pseudo code is shown in Algorithm 3.

III. IMPLEMENTATION, EVALUATION AND ANALYSIS

In this part, the effectiveness of our approach has been evaluated w.r.t accuracy and efficiency. A sample vulnerability dataset w.r.t CVE has been shown in TABLE I. This vulnerability dataset was extracted from Linux kernel and some other open source software.

Algorithm 2 Pseudo code for vulnerability detection

Input: Subject System
Output: Retrieve all vulnerable code fragments, sort them and display them w.r.t CVE-number.

```
 1: Initialize hdfs as fs, hbasetable as ht
 2: / * Consider cks1, cks2 are chunks and   ck :
    chunk, cks : chunks * /
 3: function detector(filepath)
 4: map(k, v) ⟸ filepath(k is a filepath value, v is 0)
 5: cks1 ⟸ preProcess(filePath)
 6: cks2 ⟸ chunkExtract(cks1)
 7:   for i = 0 to length(cks2) do
 8:     loop
 9:     ck ⟸ cks2.get(i)
10:     If ck is not null then
11:     scan hbase chunks, according ck's md5 hash
12:     cks ⟸ chunks
13:     end if
14:     Let result 1 is a (k, v)
15:     result1 ⟸ normalizeResult(cks)
16:     reduce(k, v) ⟸ result 1
17:       for i = 0 to length(k) do
18:         loop
19:         countChunk(v)
20:       end loop
21:     end for
22:     hf1 ⟸ reduceOutput(k, v)
23:     hf2 ⟸ locateResults(k, v)
24:   end loop
25: end for
```

Algorithm 3 Pseudo code for chunks merging

Input: sschunks: subject system chunks,
vulchunks: vulnerable chunks
Output: ssCodeSegments: subject system code segments,
vulCodeSegments: vulnerable code segments

```
 1: i ⟸ 0, j ⟸ 0
 2: ssCodeSegment ⟸ ∅, vulCodeSegment ⟸ ∅
 3: for i ⟸ 1 to vulChunk.size() − 1
 4:   for j ⟸ 1 to vulChubks.size() − 1
 5:   ssChunks.get(i).setMarked(1)
 6:   vulChunks.get(j).setMarked(1)
 7:     for i ⟸ 0 to ssChunks.size() − 1
 8:       while (ssChunks.get(i).getMarked() == 1) do
 9:           i ⟸ i + 1
10:       end while
11:       if (previousIndex + chunkSize >= i)
12:       setMarked(ssChunks, previousIndex+1, i−1)
13:     end for
14:   end for
15: end for
```

TABLE I: Vulnerability Dataset

Project Name	Number of CVE's	Number of hunks	LOC
Linux Kernel	1,025	3,935	252,229

TABLE II: Detection Results of Different Subject Systems against HBase Fingerprint Index (vulnerability source code)

Subject System	Total Files	LOC	Vulnerable Fragments
Linux 4.0	49,617	19,385,334	567
OpenSSL 1.0.2	18,611	1,303,157	197
Apache Server 2.4.1	3,909	1,072,436	79
Cinder 0.8.0	2,955	1,180,935	30
PostgreSQL 8.0	1,906	1,332,103	49
OpenCV 3.1.0	2,379	1,141,501	53
Arduino 2.3.0	680	277,710	18

System Specification: VCIPR vulnerable detection approach was performed on Linux operating system. A single machine (Intel core-i7, 3.60GHz*8 & 24 GB of RAM) was used to perform all tasks including downloading patch files, retrieving vulnerable source code, preprocessing, normalizing, fingerprint indexing and vulnerability detection.

A. Repair Patterns in Patches

RQ1: Are all the repair patterns in patch files designed to remove the vulnerability issue?
No, not all the repair patterns in patch files are designed to solve the vulnerability issue. During the experiments on patch files, we found different more common repair patterns, which are shown in Fig 4. So, for hackers, it is a very easy task to concentrate on the related patterns to attack. This analysis can also help developers to keep in mind these coding standards during the development of any system. For example, a variant of repair pattern the *Conditional Block* is the Conditional *if*

block with the addition of exception throwing to just improve the code. So, it must be clear that not all the repair patterns are considered as vulnerable code. Because some patches are just updates to improve the system and some are just updates of different drivers. That's why we used CVE's reference number to evaluate the specific vulnerability related patch files.

B. Unpatched (Vulnerable) code Detection

RQ2: Is it possible to detect the unpatched (vulnerable) code in open source systems?
Yes, It is possible to detect the unpatched vulnerable code fragments in open source systems. In this experiment, we detected all unpatched source code fragments in different open source software, which are considered as vulnerable code fragments because of CVE reference numbers. We performed experiments on seven different open source software as shown in TABLE II. Although, these detected vulnerable fragments in TABLE II are part of the already published vulnerabilities but similarly we can detect the vulnerabilities in some other related systems in which the vulnerabilities are not announced yet.

The experiment shows that there are number of similar buggy code segments that exist in these code bases, which

make them vulnerable. We have verified the presence of reported unpatched code fragments, i.e., similar exact same buggy code, to make sure VCIPR implementation is correct. Furthermore, We can download more patch files and their original vulnerable source code against CVE numbers to extend our approach to large-scale level by generating their fingerprint. By doing this, it will be very helpful to detect buggy code fragments at large-scale level. Meanwhile, it can be applied to detect unpatched code fragments in day-to-day development.

C. Unpatched (Non-vulnerable) Code Detection

RQ3: Do the unpatched (non-vulnerable) code clones affects the security measures?

Yes, the code clones, which are not even vulnerable also affects the security measures. As we discussed earlier, unpatched (vulnerable) code detection is very important to make system secure, meanwhile if a code is similar or copied from other software but not vulnerable is also a threat in security. These kind of similar code fragments are called *code clones*. They occur when a software programmer copies code from other software and reuse them even without making change in it, which is very common in software development activities. Recently, there are many approaches that perform code clone detection in large scale systems [14] [15] [16] [17] [18] [19] [7] [20] [21] [22] [23] [13], which detects clones on different level of granularities and different types of clones.

The cloned code is actually the similar source code fragment between two software or applications, where cloned type describes the degree of similarity between code fragments. Type-1 clones are the totally similar code fragments with almost 100% similarity because the code is fully identical. Type-2 have the similar source code with almost 90% similarity because in type-2 clones there are little variations in literals, names of variables, methods, and functions. In case of type-3 clones, where new lines added in the original source code or maybe deleted, it is still suspected code which can have the similarity of 60% to 80% between the source code fragments.

By using our technique VCIPR, we have also detected code clones but of type-1, because our technique is to detect vulnerable code fragments which are totally have the same features with the fingerprint value. So, there was no need to detect type-2 or type-3 clones.

D. Security Related Bugs

The security-related bugs are very important to be fixed as early as possible. When these bugs are fixed in the projects, then mostly all the relevant similar code fragments should automatically be corrected. Unfortunately, sometimes code reuse (code clones) among different open source projects makes it very difficult to update the relevant projects, mostly when a patch is released.

For an attacker it's quite easy to identify the known vulnerabilities in different projects, those are not patched yet. TABLE II shows that there are almost 567 security-related bugs have been found in Linux 4.0 version, 197 in OpenSSL,

79 in Apache, 30 in cinder, 49 in PostgreSQL, 53 in OpenCV and 18 in Arduino. These vulnerable code fragments can be detected more precisely by extending the fingerprint indexing dataset.

In this section, we show an example of a real bug we have found in Linux 4.0. The CVE-2018-1066[7] is a vulnerability in Linux kernel version before v-4.11, that allows an attacker to get control on a CIFS server. This vulnerability was published in February 2018 in NVD[8]. So, to maintain the security and get safe from hackers, this patch [24] should be applied.

E. Scalability and Efficiency

RQ4: Is the scalability an issue in vulnerability detection? Yes, scalability is becoming a big issue in code clone detection and vulnerability detection because the code is increasing day by day. The execution time actually scales with the size of the processed code (LOC). In the case of Hadoop file system, there are very fewer chances of failure, even for large-scale vulnerable code dataset, because MapReduce divides the task into subtasks. To focus on scalability we used different datasets from 1 KLOC to 1 BLOC, by selecting some open source software. Our approach can handle billion of LOC without any problem, which is quite good as compared to previous approaches. In our experiment, we find all unpatched (vulnerable) code fragments in Linux 4.0, which consists of almost 20 million lines of code. We tested our approach in both ways by using Linux 4.0 as a subject system and also using it as a fingerprint. However, scalability is not an issue in our approach, we can extend our system by just building fingerprint index of day-to-day published CVEs.

In the evaluation of scalability over LOC, TABLE III shows that our technique overwhelmed over others. DECKARD had very least scalability, which is failing to process 100 MLOC, because of a memory error. In the case of CCFinderX, an error occurred of file I/O after 3 days execution for 1 BLOC. VUDDY finished the 1 BLOC task in 14 hours and 17 minutes. But in our technique, VCIPR finishes the same task only in 12 hours and 35 minutes. Although VUDDY, SourcererCC and ReDeBug have been also scaled to 1 BLOC, but their execution time is slower than VCIPR. It can be seen from the results that our approach takes considerably less time as compared to other techniques for a big amount of LOC and there are very fewer chances of error occurrence.

The proposed approach has also made great progress in the programming language extensions that can accurately analyse the vulnerable code fragments in different languages, i.e Java, C/C++, JavaScript, Python, Php, C#, Vb and Cobol.

F. Lower False Detection Rate

VCIPR focuses on decreasing the false detection rate, by using an exact matching of hash value instead of fuzzier matching or machine learning matching techniques. So, by doing this, we may find fewer unpatched code fragments, but we will have the less false positive rate and a very low

---

[7]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1066
[8]https://nvd.nist.gov/vuln/detail/CVE-2018-1066

TABLE III: Execution Time (or Failure Condition) Comparison with other Techniques

LOC	VCIPR	VUDDY [13]	SourcererCC [25]	ReDeBug [6]	CCFinderX [26]	DECKARD [27]
1 K	1s	0.44s	2.3s	35.6s	6s	1s
100 K	9s	5.17s	50.7s	42s	50s	13s
1 M	1m 45s	55s	1m 44s	1m 43s	6m 44s	2m 20s
10 M	7m 33s	12m 43s	24m 38s	18m 32s	1h 36s	12hr 30m
100 M	1h 21m	1h 32m	9hr 42m	2hr 32m	12h 44m	Memory Error
1 B	12h 35m	14h 17m	25d 3hr	1d 3hr	File 1/O error	–

false negative rate. There are some clone detection technique as mentioned and referenced in section III-C, which could be used for vulnerability detection, but some of them don't support exact similar code fragment detection. Because they usually detect type-2 and type-3 clones also. Which mean that the little bit change in source code will also be detected as a vulnerable code, that is the symbol of increasing false rate during detection because this small change in code can be a patched code.

Our experiment shows that anybody of the field of computer science who has the little bit knowledge of hacking can find thousands of vulnerable software, among millions of lines of code in only few minutes by using a single laptop, once the patch is released.

G. Accuracy Evaluation

By extracting the vulnerable code fragments from patch file against it's CVE-reference number, we collected almost 4,000 vulnerable code fragments. We evaluate the accuracy by using standard metrics such as *TP*, *FP* and *accuracy*. We selected randomly some results for the evaluation of our technique and count the accuracy measures as shown in TABLE IV. To overcome the bias, the results were evaluated by 3 judges from our lab, who were having the knowledge of vulnerability source code. TABLE IV shows *TP*, *FP* and *accuracy* against different selected subject systems. In total, 200 files were evaluated from the selected subject system. From TABLE IV, we can see that the accuracy rate is very high for Linux subject system because most of the CVE vulnerable code belongs to Linux kernel, which was used as fingerprint index. The accuracy in all other subject system is almost the same in average of 83%.

False Positive Rate: It is a false alarm, which indicates that a vulnerability exits, when it does not. As we mentioned earlier that our vulnerability detection technique is based on code clone detection method. So, we detected all the code fragments which have 100% similarity. The reason false positive rate hit to the maximum range of 22%, because some of the vulnerabilities consist on multiple code fragments. So, whenever a similar code fragment detected, our technique considered it as a vulnerability, which can be in actually a part of vulnerability. But this problem can be solved by further evaluating these vulnerable code fragments in detail.

False Negative Rate: The term false negative mean that an approach does not detect vulnerability while it exists in the source code. As we could not have all the collection of

TABLE IV: Detected results evaluation

Subject System	Evaluated Results	True Positive	False Positive	Accuracy
Linux 4.0	100	92.2 %	8 %	92 %
OpenSSL 1.0.2	50	80.7 %	20 %	81 %
Apache Server 2.4.1	20	82.3 %	18 %	82 %
Cinder 0.8.0	10	78.8 %	22 %	79 %
PostgreSQL 8.0	10	79.5 %	21 %	80 %

known vulnerabilities, which are almost 111,914[9] in numbers upto current date. So, used an unbiased way to count false negative by adding some vulnerable code fragments into some already used open source software. These vulnerable code fragments were randomly selected from our collected dataset. By doing this it was quite easy to count false negative rate because the number of vulnerability code fragments were already known. To perform this experiment, 100 different vulnerability fragments were added into 5 selected open source software including Linux 4.0, OpenSSL 1.0.2, Apache Server 2.4.1, Cinder 0.8.0 and PostgreSQL 8.0. Then the Fingerprint index was built and the randomly added 100 vulnerability code fragments were used as a subject system. The false negative rate was counted by evaluating the retrieved results against these vulnerable code fragments. The false negative rate was quite low in average, i.e. Linux 4.0 (2%), OpenSSL 1.0.2 (2%), Apache Server 2.4.1 (1%), Cinder 0.8.0 (2%), and PostgreSQL 8.0 (1%) respectively as shown in TABLE V. The main reason, why false negative rate was too low because we detected exact similar functions in the source code, which were having 100% similarity. TABLE V shows the effectiveness and accuracy of our approach for the detection of vulnerability at function level granularity.

Furthermore, the accuracy of our vulnerability detection technique also depends on the collection of know vulnerabilities with respect to its CVE id. Meanwhile, the vulnerable code extraction by tracing and locating the patch files. As long as, the extracted vulnerable code is correct, the accuracy of vulnerability detection will automatically be high because we have used MD5 hash values against the source code. So, there are very less chances of collision in hashes.

IV. DISCUSSION

In this section, we perform a detailed discussion on the main idea and importance of this paper including performance, accuracy, efficiency, scalability, and a comparative study with the state-of-the-art-tools. First of all the main motivation

---

[9]https://cve.mitre.org/index.html

TABLE V: False negative rate

Subject System	Vulnerable Fragments (Added)	False Negative Rate
Linux 4.0	20	2 %
OpenSSL 1.0.2	20	2 %
Apache Server 2.4.1	20	1 %
Cinder 0.8.0	20	2 %
PostgreSQL 8.0	20	1 %

behind this research is to aware the computer users to the security flaws in software so that they can update or apply patches to them. Meanwhile, for the developers to let them be careful whenever they copy the code from unknown sources without making it sure that it has not a vulnerable code or whenever they use third-party components. This paper is proof of hacking perspective when a patch is released, through which the hacker can easily trace the source code and can use it for hacking purposes or to exploit it.

To prove the idea of this paper, we proposed and developed a vulnerability detection technique at function level granularity, to helps us in detecting similar code fragments in different subject systems as mentioned in TABLE II. Our approach provides a low false positive rate and very low false negative rate because we considered the exact hash matching mechanism to retrieve the results. Meanwhile, our approach support vulnerability detection in large-scale system and also support multiple languages including C/C++, Java, JavaScript, Python etc. The proposed approach is based on MapReduce, Hadoop and HBase framework, which can be extended to large scale level and can perform distributing vulnerability detection. The vulnerability source code was collected through the CVE (Common Vulnerabilities and Exposures)[10] system, by tracing the specific patch links to the source code.

As this technique provides the low false detection rate, meanwhile it performs better than other vulnerability detection approaches. It has been shown that our approach can be scaled up to billion lines of code without any failure. It has been seen that the execution time for large-scale systems is better than the small scale systems. If we talk about scalability the VCIPR is more scalable as compare to other state-of-the-art-tools as shown in TABLE III. DECKARD [27] is a tree-based code clone detection approach, which claims to be a scalable approach by representing similar subtrees of source code. This technique is based on a novel characterization of subtrees and it uses an efficient algorithm to cluster the vectors w.r.t. the Euclidean distance metric. This technique is also language-independent like ours. But as we can see in TABLE III, that when the lines of code hit 100 million, it crashes down by memory error. CCFinderX [26] is also another clone detection technique, which use token level granularity to detect clones including type-3 clones. This technique has high false positive during vulnerability detection because in case of detecting type-3 clones, it detects the changes made in the source code including addition of lines and deletion of lines. These changes might be applied to remove the vulnerability, so it has high

[10]https://cve.mitre.org

false positive rate by detecting type-3 clones. Secondly, when the code reaches to 1 billion, it crashes down to File I/O error. ReDeBug [6] is a syntax-based approach which detects un-patched source code in OS-distribution. It processed almost 2.1 billion lines of code on almost 700,000 Loc/min to build a source code data-set. So, this approach is considered to be scalable but its execution time is slower than VCIPR. It can process almost 1 billion source code in 1 day and 3 hours, but VCIPR can handle the same amount of code in 12 hours and 35 minutes. SourcererCC [25] uses a bag of tokens as granularity and it required almost 25 days and 3 hours to process 1 billion lines of code, which is quite slow but it has the high accuracy rate for detecting type-3 clones. But in the case of vulnerability detection to detect type-3 clones are not necessary at all because they will bring high false positive rate. Recently, VUDDY [13] is the vulnerability detection technique with high speed and accuracy. It is also a function level granularity technique but it ignores the short functions which are claimed as not to be vulnerable. But during our study of patch files, we saw that there are some small functions with just a few lines of code are also vulnerable and they do have the impact on the source code. VUDDY [13] can process 1 billion lines of code in almost 14 hours and 17 minutes, our technique is just a bit faster with an average time of 12 hours and 35 minutes to process 1 billion LOC. So, in overall discussion, the proposed approach is more scalable to detect the vulnerable source code.

In the case of accuracy, most of the approaches uses clone detection techniques for vulnerability detection. Unfortunately, these approaches are not suitable for vulnerability detection because they detect type-1, type-2, type-3 and even some approaches claim to detect type-4 clones, which means to say that the detected clone fragments are not 100% similar. So, these approaches also detect the changes made in the source code, and these changes can be the applied patch files to the source code, which can be indication of high false positive rate. But our proposed approach detects exact (type-1) similar code fragments based on its MD5 hash value. This study also reveals that vulnerabilities found in code clones have very high severity (risk) as compared to non-cloned code.

Furthermore, in this section, we discuss now an experimental result, in the form of vulnerability, which was detected in one of our subject system *Linux 4.0*. We detected the vulnerability refereed as CVE-2018-12714 in the file path *Linux 4.0/kernel/trace/trace_events_filter.c*. This vulnerability is a very high-risk vulnerability with a cvss base score of 10, which allows an attacker to perform a denial of services attack. The filter parsing in this file could be called as no filter, which is N=0. This attack has been fixed later on but if you are still using any version before 4.17, you must have this vulnerability in your system, so you must upgrade it or apply the available patch to it. Patch of this vulnerability made changes in two functions by deleting 1 line and adding almost eight lines of code. Although this patch has been released a few months ago but the old versions of Linux are still not patched.

V. RELATED WORK

Most of the recent work has focused on detecting vulnerabilities in open source software [3] [12], meanwhile proposing the vulnerability prediction models, which does not concern with the source code but only on previously published CVE data [4] [28] [29] [30]. Detecting code clones in code bases have been very useful in open source systems [12] [31] [32] [33]. Prior research has shown that there are many cloned and fake applications in android markets [34] [35] [36] [37]. These code clone detection techniques have high false rate because they detect code clones at the different level of granularities to detect different types of code clones. For example CCFinder [26] uses lexing and then transformations performed based on rules to detect whether the code is similar. These rules are language depended. There are very few approaches which study patch files, retrieve vulnerable code from the source code and then detect unpatched (vulnerable) code fragments from other open source software. VFDETECT [38] proposed an approach based on innovative fingerprint model to detect vulnerable code. ReDeBug [6] takes a set of LOC as it's processing unit and it slides a window $n = 4$ through the source and applies 3 hash functions to each window. The clones between files detected by bloom filter, which save the hash of each window. ReDeBug detects some of the type-3 clones, but it cannot detect type-2 clones. Consequently, it misses many vulnerable code clones, which are slightly modified. DECKARD [27] builds an ASTs for each source code file, then extracts all the characteristic vectors from ASTs. After clustering the vectors on the basis of their Euclidean distance are identified as clones. This approach requires extensive time, as the subgraph isomorphism is the time consuming problem. Meanwhile, DECKARD does not guarantee the sufficient scalability. Moreover, it was mentioned in [39] that DECKARD has a 90% false positive rate. VulPecker [40] is an automatically vulnerability checking system. It characterizes the vulnerability with the different set of features and selects one code similarity algorithms, which is optimal for vulnerable type code fragment. It detected almost 40 vulnerabilities, not registered in the NVD. But it's still improper to be used for massive open source software because it's quite slow.

VI. CONCLUSION AND FUTURE WORK

In this paper, we proposed VCIPR, which is an approach that shows the hackers perspective to find security flaws in software. The design of this approach is directed to overcome the scalability problem and detection of exact vulnerable code fragments at function level granularity. We implemented and evaluate our approach by collecting vulnerable code fragments from patch files against different CVEs. We detected and evaluated most common security pattern changes in the patch files, meanwhile we detected unpatched (vulnerable) source code in different open source software, which is a symbol of risk. VCIPR can also detect code clones in other software as well. The results show that our approach is quite efficient, reliable and scalable. Meanwhile, this paper shows that vulnerable functions always remain unfixed for many years and they propagate to other systems gradually. This paper has also a learning point for software developers to make sure the security standard during the development process. Our approach is incrementable and can also be extended to distributed vulnerability code detection, where we can build fingerprint index of a large amount of vulnerable code fragments. For future concerns, we are planning to extend our approach for vulnerability detection at file level granularity and collect a lot of vulnerable code fragments against multiple languages to generate a big vulnerable code dataset. Moreover, on the basis of our outcomes, we will build a vulnerability prediction model, which will detect vulnerable patterns in advance to overcome the security threats.

REFERENCES

[1] J. Avery and J. R. Wallrabenstein, "Formally modeling deceptive patches using a game-based approach," *Computers & Security*, vol. 75, pp. 182–190, 2018.

[2] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 143–157.

[3] O. H. Alhazmi, Y. K. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Computers & Security*, vol. 26, no. 3, pp. 219–228, 2007.

[4] A. Gorbenko, A. Romanovsky, O. Tarasyuk, and O. Biloborodov, "Experience report: Study of vulnerabilities of enterprise operating systems," in *Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on*. IEEE, 2017, pp. 205–215.

[5] H. Guo, Y.-Y. Wang, Z.-L. Pan, and S.-W. Liu, "Research on detecting windows vulnerabilities based on security patch comparison," in *Instrumentation & Measurement, Computer, Communication and Control (IMCCC), 2016 Sixth International Conference on*. IEEE, 2016, pp. 366–369.

[6] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 48–62.

[7] M. Mondal, C. K. Roy, and K. A. Schneider, "Identifying code clones having high possibilities of containing bugs," in *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 2017, pp. 99–109.

[8] Y. Y. Ng, H. Zhou, Z. Ji, H. Luo, and Y. Dong, "Which android app store can be trusted in china?" in *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*. IEEE, 2014, pp. 509–518.

[9] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 485–495.

[10] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software," in *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*. IEEE, 2008, pp. 81–90.

[11] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE, 2008, pp. 172–181.

[12] A. Sheneamer and J. Kalita, "A survey of software clone detection techniques," *International Journal of Computer Applications*, pp. 0975–8887, 2016.

[13] 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.

[14] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "Assessing code smell interest probability: A case study," 2017.

[15] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with cloneworks," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 27–30.

[16] T. Matsushita and I. Sasano, "Detecting code clones with gaps by function applications," in *2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2017*. Association for Computing Machinery, Inc, 2017.

[17] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara, "Code relatives: Detecting similarly behaving software," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 702–714.

[18] F. Al-Omari and C. K. Roy, "Is code cloning in games really different?" in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016, pp. 1512–1519.

[19] H. Keuning, B. Heeren, and J. Jeuring, "Code quality issues in student programs," 2017.

[20] T. Hatano and A. Matsuo, "Removing code clones from industrial systems using compiler directives," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, May 2017, pp. 336–345.

[21] D. E. Krutz and M. Mirakhorl, "Architectural clones: toward tactical code reuse," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016, pp. 1480–1485.

[22] D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study," *Information and Software Technology*, 2017.

[23] M. Mondal, C. K. Roy, and K. A. Schneider, "Does cloned code increase maintenance effort?" in *Software Clones (IWSC), 2017 IEEE 11th International Workshop on*. IEEE, 2017, pp. 1–7.

[24] L. Torvalds, "CIFS: Enable encryption during session setup phase," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit-, [-/?id=cabfb3680f78981d26c078a26e5c748531257ebb].

[25] V. Saini, H. Sajnani, J. Kim, and C. Lopes, "Sourcerercc and sourcerercc-i: tools to detect clones in batch mode and during software development," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 597–600.

[26] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[27] 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.

[28] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE transactions on software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.

[29] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The attack of the clones: A study of the impact of shared code on vulnerability patching," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 692–708.

[30] S. Kim and H. Lee, "Software systems at risk: An empirical study of cloned vulnerabilities in practice," *Computers & Security*, 2018.

[31] E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective-a workbench for clone detection research," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 603–606.

[32] J. Akram, Z. Shi, M. Mumtaz, and P. Luo, "Dccd: An efficient and scalable distributed code clone detection technique for big code," in *PROCEEDINGS SEKE 2018, The 30 th International Conference on Software Engineering and Knowledge Engineering*. KSI Research Inc, 2018, pp. 354–359.

[33] H. Sajnani, V. Saini, and C. Lopes, "A parallel and efficient approach to large scale clone detection," *Journal of Software: Evolution and Process*, vol. 27, no. 6, pp. 402–429, 2015.

[34] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 175–186.

[35] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 95–109.

[36] J. Akram, Z. Shi, M. Mumtaz, and P. Luo, "Droidcc: A scalable clone detection approach for android applications to detect similarity at source code level," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 2018, pp. 100–105.

[37] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM, 2012, pp. 317–326.

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

[39] G. Succi, J. Paulson, and A. Eberlein, "Preliminary results from an empirical study on the growth of open source and commercial software products," in *EDSER-3 Workshop*, 2001, pp. 14–15.

[40] 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.