IET Information Security

Research Article

IET Journals
The Institution of Engineering and Technology

How to build a vulnerability benchmark to overcome cyber security attacks

ISSN 1751-8709
Received on 18th December 2018
Revised 14th August 2019
Accepted on 3rd September 2019
E-First on 26th September 2019
doi: 10.1049/iet-ifs.2018.5647
www.ietdl.org

Junaid Akram[1] ✉, *Luo Ping*[1]

[1]*Key Laboratory of Information System Security, School of Software, Tsinghua University China, Beijing, People's Republic of China*
✉ *E-mail: znd15@mails.tsinghua.edu.cn*

Abstract: Cybercrimes are on a dramatic rise worldwide. The crime rate is growing day by day in every field or department which is directly or indirectly connected to the internet including Government, business or any individual. The main objective of this study is to evaluate the vulnerabilities in different software systems at the source code level by tracing their patch files. The authors have collected the source code of different types of vulnerabilities at a different level of granularities. They have proposed different ways to collect or trace the vulnerability code, which can be very helpful for security experts, organisations and software developers to maintain security measures. By following their proposed method, you can build your own vulnerability data-set and can detect vulnerabilities in any system by using suitable code clone detection technique. The study also includes a discussion of reasons for the rise in cybercrimes including zero-day exploits. A case study has been discussed with results and research questions to show the effectiveness of this study. This study concludes with the effective key findings of published and non-published vulnerabilities and the ways to prevent from different security attacks to overcome cybercrimes.

1 Introduction

Things are changing very fast in the world of cybersecurity, new threats are appearing on a daily basis and attackers (hackers) continuously trying their techniques to attack. Cyber security encompasses some issues from cybercrimes, including hacker breaking into a computer, perform a denial of service (DoS), execute vulnerable code, stealing personal information etc. Cybercrimes are becoming a real threat these days but it is quite different from old crimes, including robbing, mugging or stealing. Unlike these kinds of crimes, cybercrimes do not require the physical presence of hacker (criminals). These crimes can be committed remotely and the criminals do not worry about the laws, where they commit the crimes. The systems, which help people to conduct online transactions and e-commerce, are recently being exploited and attacked by cyber criminals. One very important thing in previous years has taught us is that cybercrime is one of the big issues that should not be ignored.

Vulnerability detection has improved the ability to discover the vulnerable code. A bug, CVE-2017-5638, which was revealed in March 2017, but the hackers were using it till late May 2017, against Equifax [https://www.engadget.com/2017/09/13/equifax-apache-argentina/] and they were stealing personal information including names, birth dates, social security numbers etc., for 143 million users in the US [1]. Another bug CVE-2018-1274 released on 10 April 2018 of critical severity, which cause a DoS attack to allow unlimited resource allocation to an authenticated remote user (attacker/hacker) for parsing, which actually causes a DoS attack on CPU and memory consumption. It was fixed and a patch file was released on 17 April 2018 [https://pivotal.io/security/cve-2018-1274]. OpenSSL a heart-bleed vulnerability (CVE-2014-0160) affected many types of systems, i.e. websites, web servers, operating systems (OSs), and applications because most of the affected systems at that time either used the whole library of OpenSSL or used some part of it [2]. The CVE-2018-2628 was addressed in Oracle's critical patch update security advisory, in which a remote attacker (hacker) could easily exploit the vulnerability to fully take over on Oracle *WebLogic* server. The main reasons behind these security problems on user's sides are, software is not password protected, passwords are too easy to be guessed, wireless networks are not secured, software systems contain viruses, anti-viruses are not installed, OSs are out-of-date, applications are not patched or updated etc. Also, on developer's side, the lack of security implementation knowledge, lack of audit trail, lack of regular audits, lack of software testing skills, a fast demand of software development by clients etc. In some cases, there are many developers, who have used code clones or third party components in their systems, which can be vulnerable code files, so their systems are also at big risk. Open source software (OSS) has increased the rate of vulnerabilities. The cloned code can be a vulnerable code, which was repaired by patch file in the original source file but still unpatched in other systems [3]. An automatic exploit is possible through patch files [4]. Apply patches to vulnerable code are very important to make sure security measures [5–7]. Previous work shows that when a patch file is released, an attacker (hacker) can use this patch to trace the buggy code and create an exploit [3].

In this study, we present a method to build a vulnerability benchmark, which helps users and developers to have a deep understanding of the security flaws and weaknesses (vulnerabilities) in software systems. Meanwhile, we introduce a vulnerability benchmark, in which vulnerable source code was collected from different sources against its common vulnerabilities and exposures (CVE) number. This benchmark (vulnerability) can be used to detect similar code fragments or files into other software systems by using suitable code clone detection technique. There are many clone detection techniques [8–22], which detects code clones in different software systems at different levels of granularities. Moreover, this study shows how hackers and criminals attack your systems, which are not patched or updated. The vulnerable source code was retrieved from the patch information of different systems by tracking its unpatched source code files. During our research, we have explored that the vulnerable code or the flaw in the source code can be easily identifiable, whenever a patch file is released. So, it is very important step to find unpatched files in our systems as early as possible and update them.

For the experimental purposes, we collected almost 4750 CVEs, 6958 vulnerable files, 6290 vulnerable functions and 1930 (4285 CVEs) vulnerable components. These vulnerabilities were traced and collected from different web sources at different levels of granularities including function-level, file-level, and component-level. A part of this sample vulnerability dataset has been published online, which is publicly available on GitHub [https://github.com/

znd15]. Meanwhile, this study helps us for the better understanding of patch files and vulnerabilities including zero-day vulnerabilities. A recent trend of hacking and ways of prevention are also part of this study. In addition, we have discussed the cybercrime statistics and hacking techniques to help the reader for better understating of cybercrimes. The main concern of this study is to let an individual know that the system he is using is vulnerable or not by exploring the third party components he is using, or by detecting vulnerable code fragments in his custom code through suitable code clone detection technique. In the case of his system is vulnerable then what should he do to overcome it. Furthermore, this study gives an abstract knowledge of vulnerabilities, patch files, vulnerability trend, and hacking techniques as well as the ways to be safe from hackers. In Section 6, a case study has been performed including motivation and answered some important research questions.

Our contribution has been described by the following points.

i. Study of most reported cybercrimes and their severity.
ii. Study of common hacking techniques based on collected vulnerability source code.
iii. Study of patch files structure and the most often repair patterns in the patch files.
iv. Proposed an approach to crawling vulnerable source code (benchmark) at different levels of granularities, including function level, file level, and component level.

Here is some research questions discussed in this study:

RQ1: Why do the changes required in the source code through patch files?
RQ2: Do all the repair patterns in patch remove vulnerability issues?
RQ3: Am I (end users, developers, software engineers) vulnerable, by using components of known vulnerabilities?
RQ4: Can we (end users, developers, software engineers) use vulnerable source code as a signature set to detect vulnerabilities?

1.1 Cybercrime statistics

Globally, cybercrime was the second most reported crime in the year 2016 [https://www.pwc.com/gx/en/services/advisory/forensics/economic-crime-survey.html]. Here are some big pictures of cybercrimes which happened in the past [https://en.wikipedia.org/wiki/List_of_cyberattacks] [23–25].

- Yahoo Company revealed that in the year 2013 over a billion accounts were hacked, and in year 2014 another 500 million accounts were compromised.
- A Saudi hacker released 400,000 credit cards information, which was hacked from an Israeli sports website.
- In year 2016, it was estimated that 3.2 million debit cards were compromised in major Indian banks including, SBI, HDFC, ICICI, Axis Bank etc.
- In September 2012, IEEE was exposed by its user names and plain text passwords for almost 100,000 of its members.
- In May 2017, WannaCry attack started and declared as unprecedented in scale, which affects more than 230,000 computers in almost 150 countries.
- Equifax Inc. collects and aggregates the information of over 800 million consumers and around 88 million businesses worldwide.
- In July 2016, the WikiLeaks published documents from the Democratic National Committee email leak, which was the collection of emails stolen by Russian agency hackers and published by DC Leaks. This collection consisted of 19,252 emails, 8034 attachments from the governing body of US Democratic Party.
- In year 2014, the bit-coin exchange filed for bankrupt after 460 million dollars was stolen by hackers due to weakness in the system.
- The White House computer system was hacked in October 2014.
- A computer hacker *sl1nk* releases information of penetration in the systems of the Department of Defence (DoD), Pentagon, NSA, NASA, US Military, Navy, Space, and Naval Warfare and other government websites.

If there are no vulnerabilities, there is nothing to exploit and the attacks would be much reduced. However, vulnerabilities always exist because the software often rushed to the market, and it was developed by people, and people do make mistakes, which allow attackers to attack and compromise the systems. All an attacker need is a small weakness or a flaw in your system to exploit.

1.2 Related terms and definitions

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

Hacker: A hacker is a person, who seeks to breach the defences and exploit the weaknesses in a system or a network.
Vulnerability: It is 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, access to the flaw and capability to exploit this flaw.
CVE: It is the CVE [https://cve.mitre.org] system that provides a reference by assigning CVE-IDs for publicly known security vulnerabilities.
National vulnerability database (*NVD*): The NVD [https://nvd.nist.gov/] is the United States government standards based vulnerability management repository. It includes security references, security flaws, misconfiguration, product names, and impact metrics.
Common vulnerability scoring system (CVSS) score: CVSS is assigned a score to a vulnerability, where 0.0 represents no risk; 0.1 to 3.9 represents low risk; 4.0 to 6.9 present medium risks; 7.0 to 8.9 present high risk; and 9.0 to 10.0 present a critical risk. CVSS score 10 means that this vulnerability can cause confidentiality impact: complete, integrity impact: complete (the entire system being compromised), availability impact: complete (total shut down of the resource, access complexity: low (very little skill required to exploit), authentication: not required.
Code clones: A copied code fragment from other software systems is called a code clone.

2 Study background

2.1 Common hacking techniques

Hacker maybe motivated by a reason to attack and exploit the security, reason can be as protest, profit, stealing information, a challenge, or haired person to evaluate the security weaknesses in a system. A security exploit takes advantage of known vulnerabilities. For example, vulnerabilities in SQL, file transfer protocol, hypertext transfer protocol (HTTP), Telnet, PHP, SSH etc. There are different types of hackers, i.e. white hat, black hat, grey hat, elite hacker, script kiddie, neophyte, blue hat, hacktivist, nation state, and organised criminal gangs. The following are the most common hacking techniques recently adopted by hackers [26–29].

Vulnerability scanner: Vulnerability scanner is used to check the computers on the internet for known weaknesses. Hackers commonly use port scanners applications, which check the open ports available to attack the computer.
Finding vulnerabilities: Hackers may also try to find vulnerabilities (security flaws) manually by exploring the source code of different systems. This technique is our more concerned part of this study.
Brute-force attack: This attack is about password guessing. This method is very adaptive and fast when hacker process to check all short passwords, but in case of longer passwords, he used other methods including dictionary attacks.
Password cracking: It is a process of recovering hidden passwords from data that are stored on victim's PC, including a repeatedly try to guess the password by hand, or a dictionary, or a text file that includes many passwords.

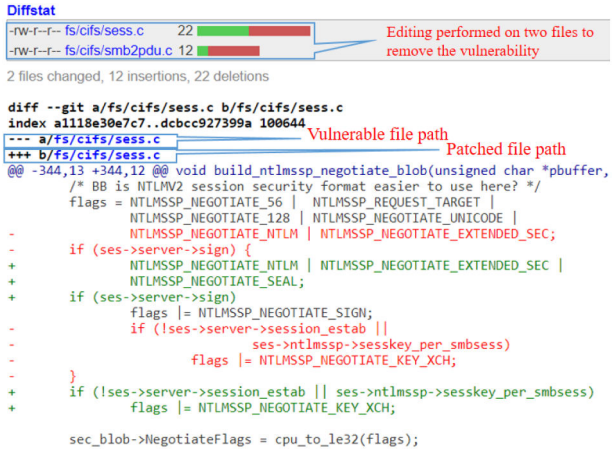


Fig. 1 *Patch 1: one hunk of a CVE-2018-1066 patch file. Linux kernel before version 4.11 is vulnerable*

Packet analyser: It is also called as a packet sniffer, which is an application used to capture data packets, passwords, and other data.

Spoofing attack (phishing): A spoofing involves a programme, an application, a system or a website that masquerades as falsifying data by fooling systems, or users to reveal the confidential information.

Rootkit: A rootkit is used as a low-level and hard to detect method to subvert the control of the OS from its operators.

Social engineering: Most of the hackers use these tactics to get enough secure information to access the company's network. The hackers may contact the system administrator and pretend as a user who cannot access his system. No one can secure an organisation if an employee of this organisation reveals a password.

Trojan horses: It is a computer programme that seems to be doing one thing but undercover it is doing another. Most of the time it is used to set up a back door, which helps the hacker to enable the intruder to gain access.

Computer virus: It is a self-replicating computer programme that helps an attacker to spread its own copies into other applications or documents.

Computer worm: It is like a virus, a self-replicating computer programme but a little bit different in working, it propagates through networks without user interaction and does not attach itself to existing programmes.

Attack patterns: Attack patterns defined as repeatable steps that applied to simulate a security attack against a system, which can be used for testing or locating vulnerabilities.

Keystroke logging: It is a tool designed to record the 'log' of every keystroke on a victim machine to retrieve them later to access the confidential information typed by the user.

2.2 Patch files and repair patterns

A patch file is a text file which actually contains the changes which are supposed to be applied in the software system and it can be in the form of an update. For an overview of the patch file, an example of CVE-2018-1066 patch file has been shown in Patch 1 (see Fig. 1). CVE-2018-1066 [https://nvd.nist.gov/vuln/detail/CVE-2018-1066] was released in February 2018, which allows hacker to control a common internet file system Server. In Patch 1, it has been mentioned that there are two files that were changed to overcome this vulnerability. So, the vulnerability was patched through a patch file, which may contain multiple diff files. However, in one diff file, there can be multiple code modifications.

Therefore, the diff file in Patch 1 shows one hunk (part of a file). The addition of lines is displayed by green colour using a symbol '+' and the deletion of lines displayed in red by using a symbol '−'. Thus, through these patch file structure, it is quite easy for a criminal to get and trace the vulnerable code to attack. By exploring and studying the commit changes in one of our article Vulnerable Code is Identifiable when a Patch is Released (VCIPR) [30], we found some very important most often repair patterns in the source code. To remove the different types of vulnerabilities from the software systems, the following repair patterns were mostly applied.

- Addition of conditional blocks
- Expression fixing
- Return value and reference change
- Change in constants
- Removing null
- Single line addition
- Wrapping/unwrapping the code

2.3 Related work

Safety is a key requirement for everyone and it can be a special challenge to face the great security threats in the field of information technology. The complexity of cyber physical systems though, render then from vulnerable and accident prone [25]. Recently, the evolution of mobile security on the banking sector and various threats including malware classification has been discussed by Wazid *et al.* [31, 32]. Srinivas *et al.* [24] have proposed some facts about the various standers of cyber security defence, architecture of cyber framework, the security threats and attacks including viruses, phishing, trojan horse, worms and DoS attacks. Eling *et al.* [33] used the peaks over threshold method to identify the cyber risks of daily life. Adams *et al.* [34] present a method to automatically ranking the attack patterns in common attack pattern enumeration and classification database, which intended to show the suggested attacks evaluated by security experts. Most of the attack patterns are ranked by the time period of attack happened. Lerums *et al.* [23] introduced a simulation model to the evaluation of effectiveness cost of cyber security options. Meanwhile, he published the findings after running the phishing attack on a workstation and revealed that the success rate on any node is almost 20%. Kamile *et al.* [35] have studied the fundamental term and conditions of cybersecurity, cyber-attacks, cyber capabilities of different countries and the cyber warfare laws.

If we talk about practically vulnerability detection, the recent work has been published on detecting vulnerabilities in android applications, binary files and OSS [5, 36–38]. Meanwhile model proposing for vulnerability prediction without considering the source code based on previously published CVE data [6, 39–41]. There are very few researchers, who study patch files, explore diff, and extract the vulnerable code to include it as vulnerability detection. Vulnerable code clone detection system based on vulnerability fingerprint (VFDETECT) [37] approach is based on fingerprint model to detect vulnerability. VulPecker [42] is a vulnerability checking system, which characterises the vulnerabilities as a set of features and uses code similarity algorithms for vulnerability detection. SourceClear [43] only scans the vulnerabilities in open source libraries; it cannot find vulnerabilities is your custom code. Our approach presents a vulnerability benchmark and introduces the ways for custom code vulnerability detection.

3 Data collection types (vulnerability dataset)

In this section, we examine the trend and rate at which the vulnerabilities occurred or discovered, and their security effects in society. The vulnerability data discussed in this study is based on the reports and information in NVD (2018) [https://nvd.nist.gov/], CVE Details (2018) [https://www.cvedetails.com/] and MITRE Corporation (2018) [https://cve.mitre.org/]. Vulnerabilities reported by the Mitre-Corporation had a standardised identification of all vulnerabilities. The candidate vulnerability is identified by reference numbers in CVE-XXXX-XXXX format. During our research we found that the following vulnerability types and their patches often appeared.

Execute code: Arbitrary code execution vulnerability is a very critical issue, which allows an attacker to execute any command on a target machine. This code execution vulnerability is most often used for the code injection. A very critical example of CVE-2016-7117 has been shown in Patch 2 (see Fig. 2). This

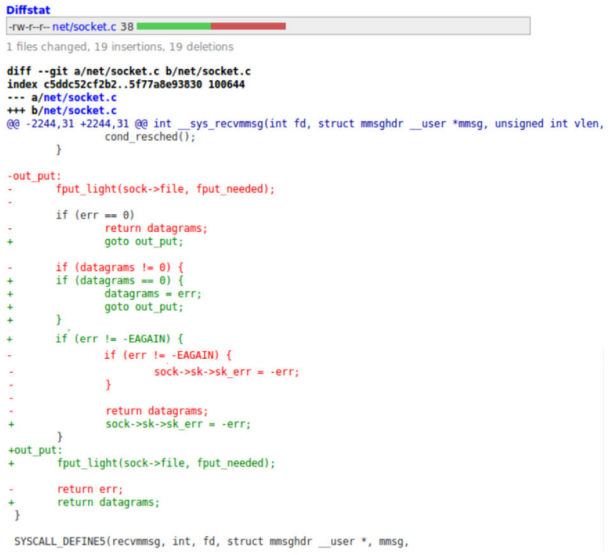


Fig. 2 *Patch 2: CVE-2016-7117 patch file to remove execute code vulnerability in Linux. CVSS score: 10 (critical risk)*

Fig. 3 *Patch 3: CVE-2016-10150 patch file to remove DoS attack. CVSS score: 10 (critical risk)*

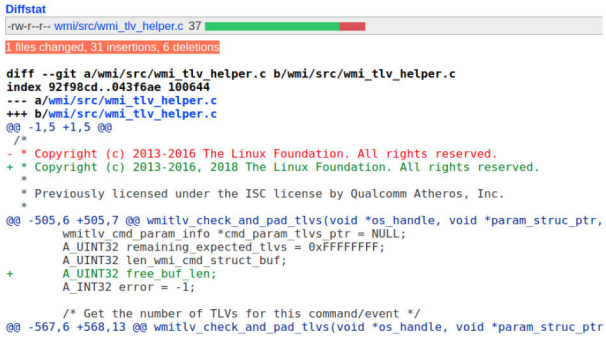


Fig. 4 *Patch 4: CVE-2018-5855 to remove buffer overflow vulnerability. CVSS score: 10 (critical risk)*

vulnerability was exploited in Ubuntu Linux (16.04), Debian Linux (7.0), and Linux Kernel (4.5.1). It has a high CVSS score rate of 10, which is the highest critical rate. This vulnerability causes a complete confidentiality impact, integrity impact, and availability impact. The hacker can totally shut down the resource very easily. If software has this vulnerability, the authentication is not required to exploit the system.

DoS: The DoS attack also called as a cyber-attack, in which the attacker seeks to make a computer machine or a network unavailable to its users. DoS is accomplished by flooding the server with superfluous requests in one attempt to overload the systems. The criminal perpetrators find this vulnerability and often target websites, services hosted web servers including banks and online payment gateways.

Linux kernel before version 4.8.13, allows attackers to cause a DoS attack or to gain privileges. Meanwhile it was also discovered that the asynchronous multi-buffer cryptographic daemon in Linux-kernel was not properly handled. The patch file has been shown in Patch 3 (see Fig. 3).

Overflow: An overflow occurs when an application tries to write data to a buffer. This error is created by storage assignments and referenced as a data type overflow. It can cause applications to crash, compromise data, and can help an attacker for privilege escalation to fully compromise on the system on which this application is running.

CVE-2018-5855 shown in Patch 4 (see Fig. 4), which was revealed and a patch was issued on 5 July 2018. It is a shrinking or padding a nested windows management instrumentation (wmi) packet in all Android (vendor: google) releases from code aurora forum (caf) using Linux kernel. This is a very critical risk oriented vulnerability, which is a part of our vulnerable code dataset. Similarly, we evaluated all the patch files against other high risk vulnerabilities types including the following.

- SQL injection
- Directory traversal
- Cross-site scripting (XSS)
- Bypass something
- Gain information
- Gain privilege
- File inclusion
- Memory corruption
- Cross-site request forgery
- HTTP response splitting

Lifetime of vulnerabilities: The number of days between the vulnerability is announced (the first versions) and their fixes (patch) being committed. So, the life time of vulnerabilities is not fixed. Many vulnerabilities remained in the source code of software systems for a long period of time, with a lifetime of 871 days, and there was a vulnerability, which was not being fixed almost 3993 days after it was introduced [44].

4 Methodology (proposed approach)

4.1 Vulnerable code extraction through patch files

In this section, we present our approach to extract vulnerability source code against its CVE reference number. There can be three ways to detect vulnerability through code clone detection techniques at different levels of granularities. So, we introduce three ways to build the benchmark, i.e. code fragment (function) level, file level, and component level. A top level view of the whole process is shown in Fig. 5.

As an example to show the method to extract vulnerable source code from software systems, CVE-2018-10074 has been shown in Patch 5 (see Fig. 6), which allows an attacker to totally shut-down the affected resource with very little effort. Meanwhile authentication is not required if an attacker wants to exploit the vulnerability. This DoS (CWE-ID: 476) attack has been found in many versions of Linux kernel, including version 1.2.0 to version 2.6.24.6, more than 700 [https://www.cvedetails.com/cve-details.php?t=1&cve_id=CVE-2018-10074] different versions were affected by this vulnerability.

The vulnerability dataset was extracted and collected by tracing, tracking and locating the changes in the patch files. The source code was collected at different levels of granularities, i.e. component-level, file-level and function-level, each of them have been further explained below.

4.2 Component-level granularity

Use of known open source vulnerable components has been increased by 120% over the last year and bundle of organisations are saying that they have no control over these components. Recently more than 1.3 million vulnerabilities are in OSS (open source components), which do not have an assigned CVE-number by NVD. More than 15,000 new or updated components are being released and available to the developers every second day. On average, many enterprises downloaded almost 170,000 Java open source components in year 2017 [https://betanews.com/2018/09/25/vulnerable-open-source-components/]. Components are

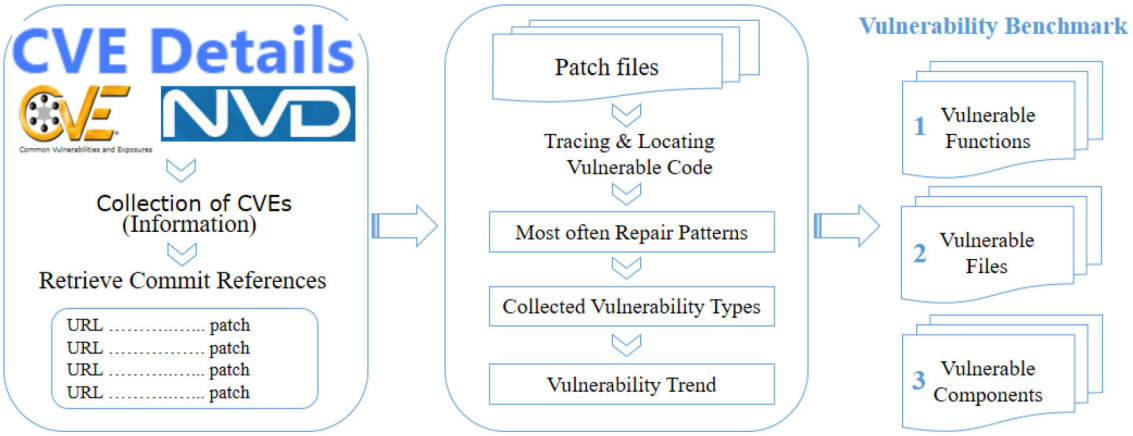


Fig. 5 *Top level view of vulnerable code extraction*

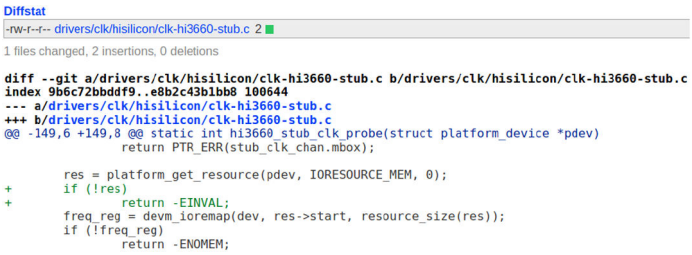


Fig. 6 *Patch 5: CVE-2018-10074: DOS*

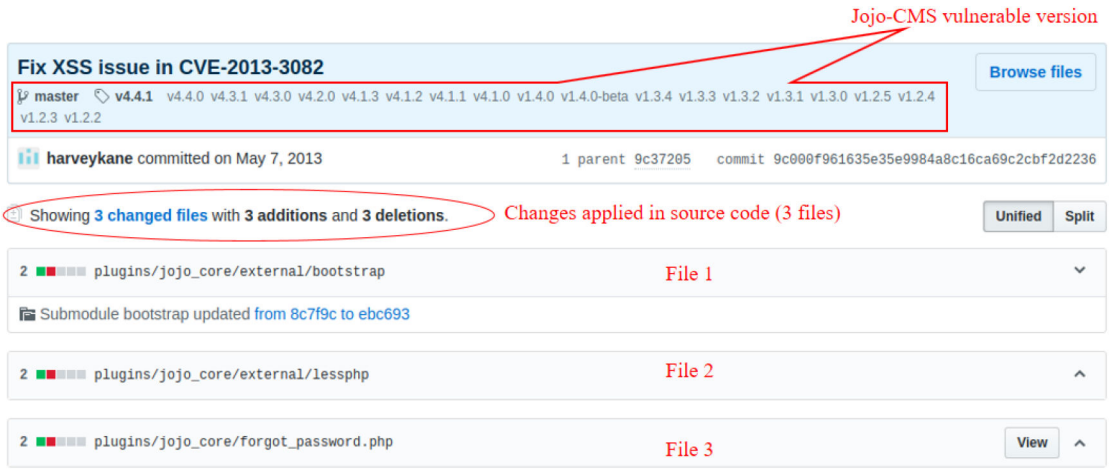


Fig. 7 *XSS vulnerability (CVE-2013-3082) in Jojo-CMS*

very essential in development of software systems, which allow and help developers to move quickly by spending less time on reinventing that project. However, this facility has increased risk. Since components such as *OpenSSL* and *Struts* are very popular in developers, at the same time these are also very attractive targets for cybercriminals.

Known vulnerability components are great threat in software systems. If an attacker knows that you are using an old version of third party component, which is not patched on your machine, then your system is at great risk. Known vulnerability components are easy to track and exploit. If one of the files in component is vulnerable and a patch has not been applied on it, then its mean that the whole component is vulnerable. The top 10 very common software components, which are outdated versions but are still in use of 90% of the time, including Curl, Dropbear, libjpeg-turbo, Expat, libjpeg, Lua, libpng Linux Kernal, and OpenSSL [https://www.synopsys.com/blogs/software-security/vulnerable-software-components/].

As an example, one of the vulnerable component (Jojo-CMS) from the dataset has been shown in Figs. 7 and 8. This is an XSS vulnerable component *plugins/jojo_core/forgot_password.php*, which allows an attacker to inject arbitrary web script via a search parameter to forgot-password. Fig. 7 shows that in this component three files have been changed by inserting and deleting some line of code. This vulnerability occurs in all versions of Jojo-CMS before version 1.2.2. During our research, we found that in some known vulnerability components, sometime it requires to change a few files to remove the vulnerability from the subject. The basic information against all vulnerability components stored in our dataset has been shown in Fig. 8. We obtained all the source code of each release against its CVE-number because a lot of software uses old version of these components. All versions are localised in GitHub repositories by mapping them with a Git commit identifiers.

4.3 File-level granularity

A patch file consists of many things including vendor name, product name, number of changes, path of original (unpatched) file, path of new (patched) file etc. So, by keep tracking the original file, which is the file before a patch applied on it, we locate this file and download it with respect to its CVE-number. As the changes applied in the source code to remove the vulnerability, all the changes should be applied in a single file or multiple files. In the case of change in the source code within one file implies the

change at file level granularity. Patch 5, shows that the changes have been applied to this file to remove the vulnerability of DoS. Meanwhile you can see that a path of original source code file has been mentioned in this patch file also, which starts with the symbols of ‘ - - - a/drivers/clk/hisilicon/clk-hi3660-stub.c’. Therefore, by following this path, we locate the vulnerable file and download it into our dataset as shown in Fig. 9. As we know that a patch of CVE-2018-10074 was applied to this file, which means that it's a vulnerable file. Similarly, we trace all the original files from different data sources against their CVE-numbers to build a vulnerability benchmark at file level granularity.

4.4 Function-level granularity

Developers usually spend time and focus on securing their own source code but forget about the code they had imported from other sources. Whenever a vulnerability published on the internet, an attacker can simply take the payload and use it against the target to exploit. In the process of collecting vulnerable functions, we performed reverse patching, which means that we retrieve the vulnerable code fragment on which the patch was applied. During function extraction, we find that some functions are not fully independent and the change in them maybe an effect of some other functions. Sometimes the changes (patches) are scattered in different functions, some of them even exist across different dependent functions in different source files. However, during our research, we saw that most of the time the vulnerable code exists only in one independent function. The patch file has a full record of the modified code including the lines to delete, the lines to add and the required changes on exact locations. So, we trace all the functions from its old (unpatched) source code file and retrieve them as a vulnerable function.

When it comes to function-level granularity, the functions containing the hunks are often included in the patch files (@ @ – line numbers); so we locate those specific line numbers and extract the concerned function from the source code. As it has been shown in Patch 1 (@ @ −344), where 344 is the line number in the source code file that has been modified. So, by tracing these line numbers we downloaded the vulnerable function before modification.

An example of vulnerable function of CVE-2018-10074 has been shown in Fig. 10. This function allows hacker to a perform DoS attack, in which authentication is not required to exploit. This vulnerability is in almost all the versions of Linux kernel before 2.6.24.6. The latest update in the form of patch file was published in May 2018. This function was extracted from the source code file of *drivers/clk/hisilicon/clk-hi3660-stub.c* shown in Fig. 9. This function starts from line number 99 and end at line 131, the patched code or the applied changes in this function was deleted to make it vulnerable. Similarly, we extract all the vulnerable function from the source code against its CVE number, meanwhile we extract the basic information about this extracted function including lines of code, changes in the source, function name, path of the original (unpatched) file etc.

5 Key findings

5.1 Vulnerability trend (cyber security attacks)

In this section, we discuss the vulnerability appearance trend (1999–2018) with respect to vulnerability type, year, and CVSS scores. Bundles of vulnerabilities in open source systems have been downloaded to create a benchmark of vulnerabilities. All the collected source code is considered as authentic because the data of these vulnerabilities were taken from two authentic websites CVE and NVD. During our research of vulnerability analysis and building a vulnerable source code benchmark, we have explored different vulnerability trends in software systems including exploit happens by vulnerability types, by vendors, by products, and by years. The dataset used for vulnerability trend analysis is almost 115 K CVE entries since year 1999 to 2018. These vulnerability trends are explained below.

Number of vulnerabilities by type: In our vulnerable source code dataset, we divide our benchmark against different types (13) of vulnerabilities as shown in Fig. 11. The largest percentage in vulnerability trend is an *execute code* vulnerability, in which an attacker can easily execute any command of his choice on the target machine. This vulnerability is the main focus of cybercriminals to attack. The number of *execute code* vulnerabilities has been already reached 30,042, which is almost double the sum of the other seven vulnerabilities shown in Fig. 11. *DoS* is the second largest published vulnerability after *execute code*. *Buffer overflow*, *XSS*, *gain information* and *sql injection* are also main focus of attackers respectively. *HTTP response splitting* vulnerability is very less focused by hackers with only 159

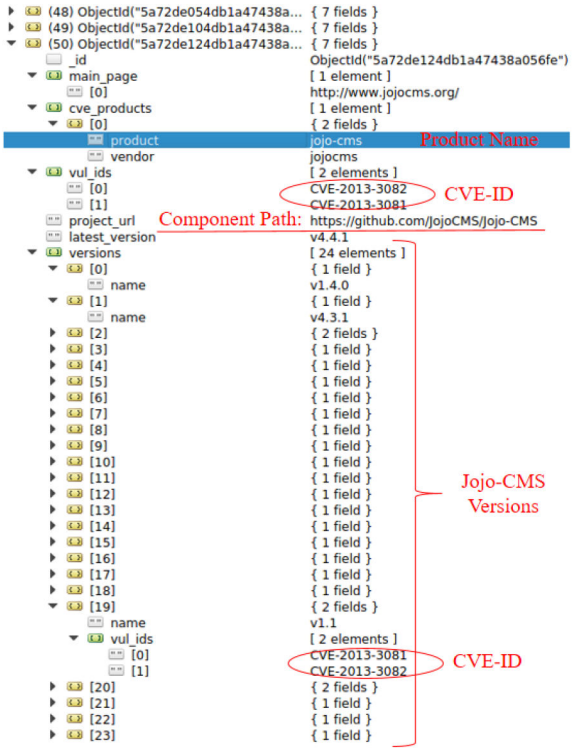
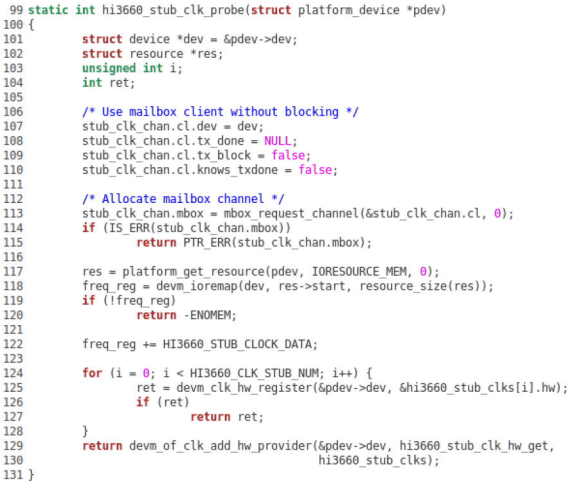


Fig. 8 *Dataset recorded entities against the vulnerability component of Jojo-CMS. Total downloaded components: 1933*

```c
#include <linux/clk-provider.h>
#include <linux/device.h>
#include <linux/err.h>
#include <linux/init.h>
#include <linux/mailbox_client.h>
#include <linux/module.h>
#include <linux/of.h>
#include <linux/platform_device.h>
#include <dt-bindings/clock/hi3660-clock.h>
#define HI3660_STUB_CLOCK_DATA (0x70)
#define MHZ (1000 * 1000)
#define DEFINE_CLK_STUB(_id, _cmd, _name) \
{ \
.id = (_id), \
.cmd = (_cmd), \
.hw.init = &(struct clk_init_data) { \
.name = #_name, \
.ops = &hi3660_stub_clk_ops, \
.num_parents = 0, \
.flags = CLK_GET_RATE_NOCACHE, \
}, \
},
#define to_stub_clk(_hw) container_of(_hw, struct hi3660_stub_clk, hw)
struct hi3660_stub_clk_chan {
        struct mbox_client cl;
        struct mbox_chan *mbox;
};
struct hi3660_stub_clk {
        unsigned int id;
        struct clk_hw hw;
        unsigned int cmd;
        unsigned int msg[8];
        unsigned int rate;
};
static void __iomem *freq_reg;
static struct hi3660_stub_clk_chan stub_clk_chan;

static unsigned long hi3660_stub_clk_recalc_rate(struct clk_hw *hw,
                                                 unsigned long parent_rate)
{
        struct hi3660_stub_clk *stub_clk = to_stub_clk(hw);

        /*
 * LPM3 writes back the CPU frequency in shared SRAM so read
 * back the frequency.
 */
        stub_clk->rate = readl(freq_reg + (stub_clk->id << 2)) * MHZ;
        return stub_clk->rate;
}

static long hi3660_stub_clk_round_rate(struct clk_hw *hw, unsigned long rate,
                                       unsigned long *prate)
{
        /*
 * LPM3 handles rate rounding so just return whatever
 * rate is requested.
 */
        return rate;
}
static int hi3660_stub_clk_set_rate(struct clk_hw *hw, unsigned long rate,
                                    unsigned long parent_rate)
{
        struct hi3660_stub_clk *stub_clk = to_stub_clk(hw);

        stub_clk->msg[0] = stub_clk->cmd;
        stub_clk->msg[1] = rate / MHZ;

        dev_dbg(stub_clk_chan.cl.dev, "set rate msg[0]=0x%x msg[1]=0x%x\n",
                stub_clk->msg[0], stub_clk->msg[1]);

        mbox_send_message(stub_clk_chan.mbox, stub_clk->msg);
        mbox_client_txdone(stub_clk_chan.mbox, 0);

        stub_clk->rate = rate;
        return 0;
}
static const struct clk_ops hi3660_stub_clk_ops = {
        .recalc_rate    = hi3660_stub_clk_recalc_rate,
        .round_rate     = hi3660_stub_clk_round_rate,
        .set_rate       = hi3660_stub_clk_set_rate,
};
static struct hi3660_stub_clk hi3660_stub_clks[HI3660_CLK_STUB_NUM] = {
        DEFINE_CLK_STUB(HI3660_CLK_STUB_CLUSTER0, 0x0001030A, "cpu-cluster.0")
        DEFINE_CLK_STUB(HI3660_CLK_STUB_CLUSTER1, 0x0002030A, "cpu-cluster.1")
        DEFINE_CLK_STUB(HI3660_CLK_STUB_GPU, 0x0003030A, "clk-g3d")
        DEFINE_CLK_STUB(HI3660_CLK_STUB_DDR, 0x00040309, "clk-ddrc")
};
static struct clk_hw *hi3660_stub_clk_hw_get(struct of_phandle_args *clkspec,
                                             void *data)
{
        unsigned int idx = clkspec->args[0];

        if (idx >= HI3660_CLK_STUB_NUM) {
                pr_err("%s: invalid index %u\n", __func__, idx);
                return ERR_PTR(-EINVAL);
        }

        return &hi3660_stub_clks[idx].hw;
}
static int hi3660_stub_clk_probe(struct platform_device *pdev)
{
        struct device *dev = &pdev->dev;
        struct resource *res;
        unsigned int i;
        int ret;

        /* Use mailbox client without blocking */
        stub_clk_chan.cl.dev = dev;
        stub_clk_chan.cl.tx_done = NULL;
        stub_clk_chan.cl.tx_block = false;
        stub_clk_chan.cl.knows_txdone = false;


        /* Allocate mailbox channel */
        stub_clk_chan.mbox = mbox_request_channel(&stub_clk_chan.cl, 0);
        if (IS_ERR(stub_clk_chan.mbox))
                return PTR_ERR(stub_clk_chan.mbox);

        res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        freq_reg = devm_ioremap(dev, res->start, resource_size(res));
        if (!freq_reg)
                return -ENOMEM;

        freq_reg += HI3660_STUB_CLOCK_DATA;

        for (i = 0; i < HI3660_CLK_STUB_NUM; i++) {
                ret = devm_clk_hw_register(&pdev->dev, &hi3660_stub_clks[i].hw);
                if (ret)
                        return ret;
        }
        return devm_of_clk_add_hw_provider(&pdev->dev, hi3660_stub_clk_hw_get,
                                           hi3660_stub_clks);
}
static const struct of_device_id hi3660_stub_clk_of_match[] = {
        { .compatible = "hisilicon,hi3660-stub-clk", },
        {}
};
static struct platform_driver hi3660_stub_clk_driver = {
        .probe  = hi3660_stub_clk_probe,
        .driver = {
                .name = "hi3660-stub-clk",
                .of_match_table = hi3660_stub_clk_of_match,
        },
};
static int __init hi3660_stub_clk_init(void)
{
        return platform_driver_register(&hi3660_stub_clk_driver);
}
subsys_initcall(hi3660_stub_clk_init);
```

Fig. 9 *Vulnerable source code file (clk-hi3660-stub.c) retrieved through the patch file of CVE-2018-10074*



```c
static int hi3660_stub_clk_probe(struct platform_device *pdev)
{
        struct device *dev = &pdev->dev;
        struct resource *res;
        unsigned int i;
        int ret;

        /* Use mailbox client without blocking */
        stub_clk_chan.cl.dev = dev;
        stub_clk_chan.cl.tx_done = NULL;
        stub_clk_chan.cl.tx_block = false;
        stub_clk_chan.cl.knows_txdone = false;

        /* Allocate mailbox channel */
        stub_clk_chan.mbox = mbox_request_channel(&stub_clk_chan.cl, 0);
        if (IS_ERR(stub_clk_chan.mbox))
                return PTR_ERR(stub_clk_chan.mbox);

        res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        freq_reg = devm_ioremap(dev, res->start, resource_size(res));
        if (!freq_reg)
                return -ENOMEM;

        freq_reg += HI3660_STUB_CLOCK_DATA;

        for (i = 0; i < HI3660_CLK_STUB_NUM; i++) {
                ret = devm_clk_hw_register(&pdev->dev, &hi3660_stub_clks[i].hw);
                if (ret)
                        return ret;
        }
        return devm_of_clk_add_hw_provider(&pdev->dev, hi3660_stub_clk_hw_get,
                                           hi3660_stub_clks);
}
```

Fig. 10 *Vulnerable function (CVE-2018-10074)*

vulnerabilities. The detailed chart of different types of vulnerabilities is shown in Fig. 11.

Number of vulnerabilities in the top 25 products: Most of the vulnerability dataset consists of *Linux Kernel* vulnerabilities, the main reason is that it is open source, so easy to trace through patch files. In chart of Fig. 12, we can see that in overall Linux kernel has more published vulnerabilities than the other products. Most of the patch files discussed in previous sections, are from Linux kernel.

Number of vulnerabilities in top 10 vendors: Top ten products with a high number of vulnerabilities are shown in the chart of Fig. 13, made up for 13 most common published vulnerabilities since 1999 to 2018. The chart shows that the products of vendor Microsoft have the highest vulnerability rate as compared to other vendors. There are almost 110,339 vulnerabilities published in last 20 years, and this rate is increasing day by day. Most of the vulnerabilities have been found in *Windows Server 2008*, which are almost 1119. *Windows 7*, *Internet Explorer*, *Windows Vista*, *Windows Server 2012*, *Windows Xp*, *Windows 8.1* and *Windows 10* are afterwards, respectively. *Apple* is almost half of *Microsoft*. Mozilla has a higher vulnerability rate than Google. Even though most of the Linux applications are open source but still they have a low

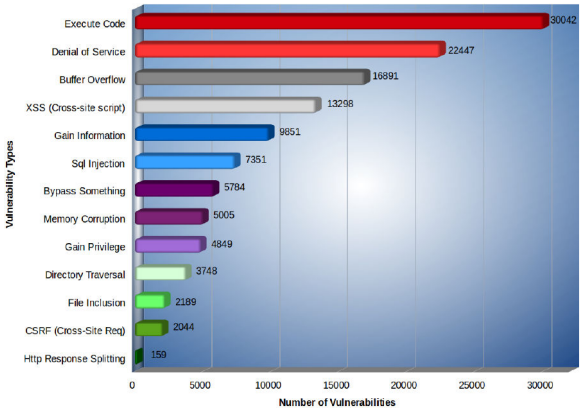


Fig. 11 *Number of vulnerabilities by type*

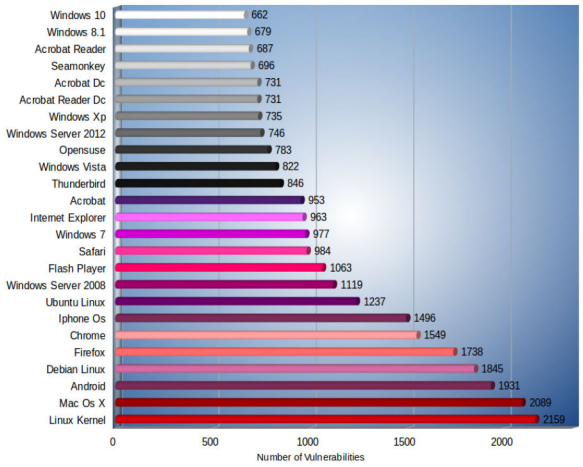


Fig. 12 *Number of vulnerabilities in top 25 products*

published vulnerability rate. The vulnerability data collected by us is based on NVD and CVE.

Number of vulnerabilities by year: As we know that the volume, scope, and cost of cybercrime are increasing day by day and these crimes are set to get even higher. Over the last ten years, cybercrimes have risen up to 20 times. There are almost 24,000 malicious mobile applications blocked everyday (Symantec). In year 2016, 3 billion Yahoo accounts were hacked (Oath.com). In year 2017, 412 million accounts were stolen from *Friendfinder's* site (Leaked Source). So, the vulnerability detection has become a very hot topic in recent years. The chart in Fig. 14 shows the known vulnerabilities of recent 11 years in different software systems and mobile applications year wise. From the chart, we can see that in year 2011, there are very few vulnerabilities as compare to other years from 2008 to 2018. It is clear from the chart that year 2017 has been a very high profile in vulnerabilities. Although, these vulnerabilities have already been published and the patches have been committed, but still they are going to have impact on years 2018 and 2019.

Distribution of vulnerabilities by CVSS scores: The CVSS provides a framework for the characteristics of vulnerabilities. It's a quantitative model, which ensures the measurement of vulnerabilities to generate the scores. The chart in Fig. 15 shows the published vulnerabilities against its severity level (0 to 10). We can see from Fig. 15, that the vulnerabilities from 7–8 CVSS scores (high severity) are the most. The severity has been categorised as none (0.0), low (0.1–3.9), medium (4.0–6.9), high (7.0–8.9) and critical (9.0–10.0).

5.2 Zero-day exploit (advanced cyber-attacks)

In previous sections, most of the time we discussed the known vulnerabilities with assigned CVE numbers. In this section, we discuss the unknown vulnerabilities or unknown exploits, which is

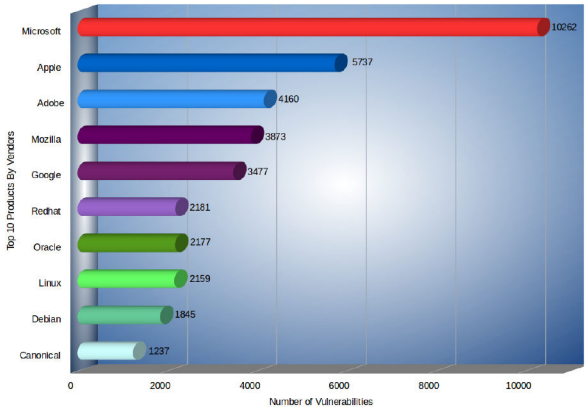


Fig. 13 *Number of vulnerabilities in top 10 products by vendor*

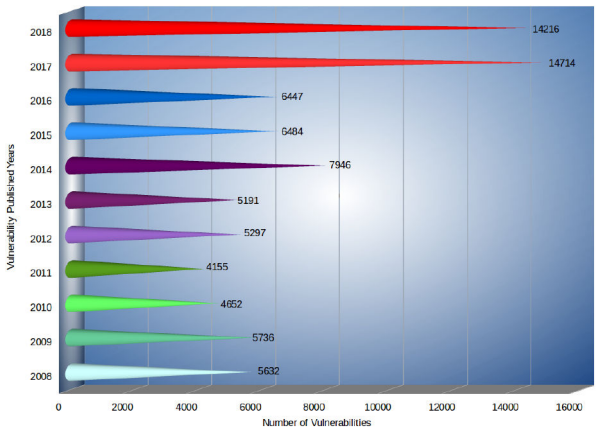


Fig. 14 *Number of vulnerabilities by year*

a complicated problem. A zero-day vulnerability leaves no opportunity for detection because no one realises that something is wrong. Once the patch is committed, the vulnerability is not called zero-day. Attackers love the zero-day vulnerabilities because none of them stop them to exploit. Some companies even pay a high amount of money if you inform them the zero-day vulnerabilities, e.g. *Rhodium* offers almost $45,000 for Linux zero-day vulnerability and up to $500,000 for encrypted applications including WhatsApp, iMessage, and Telegram.

Here we are going to explain how our vulnerability benchmark can help users to overcome on zero-day exploits. Whenever an exploit happened, we have to apply patches to our source code or to update our systems. Sometimes, if one vulnerability exploited in one system as a zero-day vulnerability, the same vulnerability can exist into some other systems but has not been found or exploited. Thus, the hacker can use this vulnerability info to cause an attack on other systems which contains the same vulnerability. We can retrieve the vulnerable code from our benchmark at the function level, file level, or component level granularity and we can detect the other vulnerable system which is using the same vulnerable code (by using code clone detection techniques) and cause an attack on them.

5.3 How to prevent

As there are thousands of published and non-published vulnerabilities but still there are many ways to prevent exploitation by removing unused dependencies, unknown files, unnecessary features, documents, components, and files. Continuously updating the versions of client side and server side components including frameworks, dependencies, and libraries using tools like Dependency-Check [https://github.com/jeremylong/DependencyCheck] etc. Dependency-Check is a tool used for software composition analysis, which detects publicly disclosed vulnerabilities within a project's dependencies by determining if there exists a common platform enumeration identifier for a given

```
3047061 Name: CVE-2018-10021
3047062 Status: Candidate
3047063 URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10021
3047064 Phase: Assigned (20180411)
3047065 Category:
3047066 Reference: MLIST:[debian-lts-announce] 20180718 [SECURITY] [DLA 1423-1] linux-4.9 new package
3047067 Reference: URL:https://lists.debian.org/debian-lts-announce/2018/07/msg00020.html
3047068 Reference: MISC:http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?
        id=318aaf34f1179b39fa9c30fa0f3288b645beee39
3047069 Reference: MISC:https://bugzilla.suse.com/show_bug.cgi?id=1089281
3047070 Reference: MISC:https://github.com/torvalds/linux/commit/318aaf34f1179b39fa9c30fa0f3288b645beee39
3047071 Reference: UBUNTU:USN-3678-1
3047072
3047073 ** DISPUTED ** drivers/scsi/libsas/sas_scsi_host.c in the Linux kernel
3047074 before 4.16 allows local users to cause a denial of service (ata qc
3047075 leak) by triggering certain failure conditions. NOTE: a third party
3047076 disputes the relevance of this report because the failure can only
3047077 occur for physically proximate attackers who unplug SAS Host Bus
3047078 Adapter cables.
3047079
3047080 Current Votes:
3047081 None (candidate not yet proposed)
```

Fig. 15 *Distribution of vulnerabilities by CVSS scores*

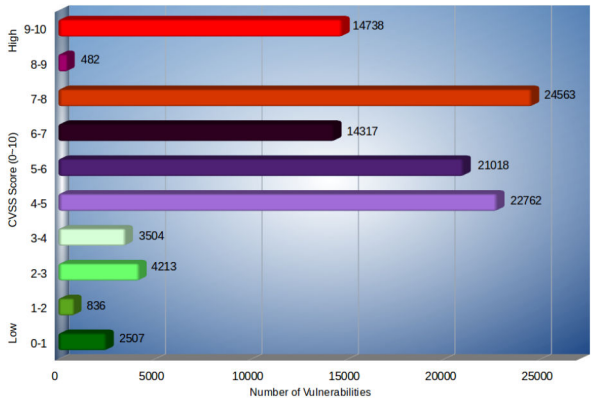


Fig. 16 *One CVE item from the CVE-collection list; In total:109,609 CVEs*

dependency. If it found this dependency, it will automatically generate a report by linking its associated CVE entries.

We can prevent vulnerabilities to be exploited, by continuously monitoring the vulnerability sources such as CVE and NVD. Use antivirus and software analysis tools to evaluate and automate software systems. Subscribe the email alerts for the security related components you use, so that you can receive update alerts on time. Only obtain the components from official sites and over a secure link. Check the hash values before you use these components and prefer the signed packages to make sure not to use malicious or modified components. Monitor all the libraries and components you use. Must apply patches and updates to your systems to overcome the security threats. As we know that the enterprises cannot stop using the open source components, which cause a great threat to your system unless the security department of your company has visibility over these components.

The vulnerable code dataset presented in this study can also be used as a fingerprint to evaluate your custom software code, by using suitable code clone detection technique [14–22, 45]. As our dataset consists of function-level, file-level, and component-level granularities, there are some clone detection techniques that help us to detect vulnerabilities on different levels of granularities. For example, a technique VCIPR [30] detects vulnerabilities at function-level granularity. Thus, it is very important to check and remove vulnerabilities from your source code as early as possible. There are some other recommendations to reduce security risks, as described below.

Keep your firewall ON: A firewall helps us to protect our computers from criminals, who try to gain access to steal the information, data, crash it, delete it, steal passwords or sensitive information. Firewalls are widely recommended for single computers.

Install or update antivirus: Antiviruses are designed to prevent malicious programmes on your computer. If it detects a virus, it works to remove it. Most of the antiviruses can be updated automatically. Furthermore, keep your system up-to-date.

Install anti-spyware: Spyware is surreptitiously installed on your machine to let others spy on your activities. Some spyware just collects information about you without letting you know. Some OSs provide free spyware protections or inexpensive software to download. So, must install anti-spyware on your system.

Keep your OS up-to-date: As we can see from the figures of the previous section, a lot of vulnerabilities have been found in different OSs. Computer OSs always publish updates to stay tune with technology and to fix security holes. So, be sure to install all the updates to make sure your computer is protected.

Be careful what you download: You have to be very careful what you download from the internet, including e-mail attachments, news offers, blogs etc. Do not open any e-mail attachment from an unknown person, these attachments may be malicious code.

Turn off your computer: Beyond the firewall protection, try to shut down your computer when you are not using it or whenever you see something is going wrong with your system, it can be possible that somebody is using your system off the screen.

Set a policy: Security teams should set a security policy, which explicitly lays that which component requires an action about vulnerability, and at what time.

Educate developers: Some developers may not have knowledge of vulnerability components or security patches. So, educate and train them about vulnerability detection techniques to make sure security measures. Also, integrate the security testing techniques in software development life cycle.

6 Case study of vulnerable code extraction

This section presents the results and the collection of vulnerable code at a different level of granularities.

Motivation: The main motivation of this research is to introduce a way of building vulnerability benchmark at a different level of

Table 1 Vulnerable source code benchmark-1

Source	Number of CVE's	Number of files	Number of functions	Language
git.kernel.org	1168	1999	3016	C/C++
github.com	1687	1970	3493	Javascript/C/C++/etc.
chromium.googlesource.com	24	58	73	Javascript/C/C++

Table 2 Vulnerability source code benchmark-2

Source	No. of CVEs	Number of files	LOC
git.videolan.org	194	202	227,802
cgit.freedesktop.org	131	225	239,009
git.qemu.org	123	138	189,458
git.savannah.gnu.org	91	116	178,409
git.openssl.org	85	130	149,565
git.ghostscript.com	79	121	123,510
total	1871	2931	3,364,586

Table 3 Vulnerable components of top 15 languages including (Java, Javascript, C, C++, Python, Html, C# etc.)

Source	Number of CVEs	Number of components
multiple web sources	4285	1933

granularities to overcome security attacks. Meanwhile, our research is very helpful for security experts, organisations, hackers and software developers to know about security threats at a deep level.

Approach: To get the vulnerable source code, we need to locate the vulnerability. A very reliable vulnerability data source (CVE-mitre) [https://cve.mitre.org] used to make sure that we are collecting the exact and authentic vulnerability source code against its CVE number and other details including name, status, URL, references, disputed etc. CVE has announced almost 109,650 known vulnerabilities with respect to its basic information. We retrieve the specific links (highlighted in the yellow of Fig. 16) to the patch files of these vulnerabilities and downloaded the vulnerable source code. Fig. 16 shows the record of one CVE, which performs a DoS attack.

Sample benchmark: To show the way of collecting vulnerable source code and to show the effectiveness of our approach, we selected different platforms, including Kernel git repository [https://git.kernel.org/] hosted at kernel.org (Singapore), GitHub [https://github.com/] repository and GoogleSource [http://chromium.googlesource.com/] etc., as shown in Tables 1–3. We extract all the patch commits and their source code against each CVE numbers. By using these commits, we locate the patch files and their original source code files and downloaded them all. With the help of diff files extract all the vulnerable function from the source code, as described in earlier Section 2.2. There are millions of commits on these platforms but we just considered that have the CVEs. During our research, we found that some vulnerabilities consist of multiple commits and some of them just one commit. In average mostly commits consist of almost three to four changes, and these changes occurred in a single file.

During the extraction of vulnerable functions, we see that some of the CVEs have more than eight hunks (changes). As an example, one of the CVE in the Linux kernel (CVE-2018-10021) [http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10021], which allows the users to cause a DoS attack. It just has one file in its commit on which the patch was applied, but we extracted six functions from this single file, which means that at six places the change has occurred to overcome this DoS attack.

A sample collection of vulnerability source code of function-level, file-level, and component-level granularity has been shown in Tables 1–3, respectively. These vulnerabilities were traced and collected from different sources by running a web-crawler with the help of the described method of Section 4.1. Table 1 shows the list of sources, number of CVEs, number of vulnerable files, and number of vulnerable functions in these files with respect to the language. Furthermore, we can see that some CVEs have multiple vulnerable files and in some vulnerable files, there are several vulnerable functions. Table 2 shows the vulnerability collection of only file-level granularity. Table 3 shows the vulnerable components with respect to its CVE number as shown in Fig. 8. A part of this sample vulnerability data-set has been published online, which is publicly available on GitHub [https://github.com/znd15].

RQ1: Why do the changes required in the source code through patch files?

The patch is a set of changes in the source code, which supports updates and fixes. Software companies create and distribute patch files that contain data, which is needed to fix a problem or update a particular application with an associated programme. Most of the fixes are vulnerabilities to remove the security flaws from the source code. The size of a patch file varies from few bytes to megabytes, meanwhile, it also depends on the changes applied to a whole file or just to a portion of a file. In the case of game applications, the patches are very large, whenever it requires to change or add graphics or sounds in it. In the case of OSs, the patches perform very important role in fixing the security holes. Some patches solve the driver issues.

In one of our research paper, VCIPR [30], we have studied many patch files in detail. We retrieved most often repair patterns in almost 2000 patch files. These all repair patterns were studied in the domain of vulnerabilities that show what kind of changes developers make in the patch files to overcome the vulnerability.

Change a file using diff: Patch reads a file that contains the output from diff, which describes the changes from an old file to a new file. The patch applies these changes to another file. Whenever we make a set of changes in a file, we use patch to incorporate these changes in some other versions of the same file. During applying patches, it saves an original copy of the file in a backup. Mostly, the backup file name remains the same as the original by adding a string prefix. In the case of the file already exists with the same name, the patch overwrites it. With the help of context format a patch can recognise when the line numbers do not match to the file being patched. In that case, patch scans forward and backward to match the exact line in that file and then proceed to apply the required changes in it. If the patch is unable to find a match, it writes that portion as a reject file.

RQ2: Do all the repair patterns in the patch file remove the vulnerabilities?

No, not all the repair patterns in patch files used to remove vulnerability because of some patches or just normal updates. Some patches include little bit variation of hot-fixes, programme temporary fixes, service pack fixes, software update fixes, slipstreaming fixes etc. Since in this study, we just have considered the patches of vulnerability fixes. That is why we collected vulnerable source code at different granularities with respect to CVE numbers. Also, our proposed method is mainly focused on already published and known vulnerabilities.

RQ3: Am I (end users, developers, software engineers) vulnerable, by using components of known vulnerabilities?

Yes, you are vulnerable if you are using known vulnerability components or the part of that vulnerable component [46, 47]. Furthermore, if you are using known vulnerable files or functions, which we have mentioned in earlier sections, you are still vulnerable. Therefore, it is quite important to update or to not include known vulnerable components. As an example, some sample vulnerable components, files, and functions have been mentioned in this study. In theory, it is quite easy for an attacker to figure out that if you are using any vulnerable library or component. Our proposed approach can help you to have a deep understanding of vulnerabilities to improve your security measures. If one of your used components have vulnerability, which is

already known and an update (patch) have been released, you should carefully evaluate it and check the source code to make sure you have applied the patch or you are not using that component.

Components typically run with the same privileges as of the application itself, therefore vulnerabilities in components can directly or indirectly cause a serious impact. These flaws can be coding errors or backdoors in components. Thus, if we include a vulnerable component, the hacker can trace us and can easily perform an exploit because he already knows the flaw in that component. For example, a CVE-2017-5638 is a Jakarta multi-part parser in *Apache Struts* before version 2.3.32, which has an incorrect exception handling during the file upload attempt. This vulnerability allows an attacker to execute the arbitrary command and it does not require any authentication to exploit the vulnerability. Attackers can exploit all the outdated versions of this to run malicious code on the server.

RQ4: Can we (end users, developers, software engineers) include vulnerable source code as a signature set to detect vulnerabilities?

Yes, it is possible to use vulnerable source code as a fingerprint to detect vulnerabilities by using suitable code clone detection technique [14–22]. However, to accomplish this mission, we are supposed to have huge number of vulnerability source code collection. Also, the collection of vulnerability source code should be authentic and proved to get high accuracy during vulnerability detection.

7 Discussion

Our approach in this study presents the ways for custom code vulnerability detection by using any available code clone detection technique. Furthermore, our study shows the cyber security trend since the CVEs have been assigned to different vulnerabilities. Then it concludes with effective key findings, which help to prevent from different security attacks.

Utilisation of dataset: This vulnerability benchmark is categorised into function-level, file-level, and component-level granularities, which can help a reader to understand the vulnerability structure, importance of patch files, and to detect vulnerabilities in their systems at the source code level. Meanwhile, they can find out which libraries or components are vulnerable in their system. Therefore, they can remove those components or upgrade to their latest versions, which are not vulnerable in their current state. In the case that their system has cloned code, which can be a piece of malware or copied from a vulnerable component, they can trace and locate this code fragment through available clone detection techniques.

Dataset collection limitations: In this study, we mainly focused on vulnerabilities collection in OSS and components. Vulnerabilities are increasing day by day, up to now there are almost 115 K published vulnerabilities, including open source and closed source. As long as it concerned to open source vulnerabilities, in which patch files are easily available and we can locate the vulnerable code to download it. However, in the case of closed source, it is hard to locate its source code, including vulnerabilities in Internet Explorer (Microsoft), MS Office (Microsoft), Windows (Microsoft), Oracle etc. Thus, we collected vulnerabilities in OSS. We are still collecting vulnerable source code fragments, files, and components to make a big benchmark for developers to facilitate them to check their source code time to time.

Dataset accuracy: This study provides a benchmark of vulnerable source code and vulnerable components. As long as it concerns to the accuracy of our collected vulnerable dataset, it is very accurate because we extracted all the vulnerabilities on the basis of its CVE numbers, which were assigned by very authentic sources, i.e. CVE and NVD. If a developer uses a library or any component which is vulnerable, indirectly he is vulnerable. Since hacker can trace him and he can attack his system through that library. Thus, it is very important, to never include that vulnerable library, unless it must be patched or should be the latest version. Simply matching against CVEs will not lead to false positive, because there are many clone detection techniques, which provide 99% accuracy to detect similar code fragments (type-1 code clone detection). As it has been mentioned that there are some dependencies in the functions to detect vulnerability at function-level granularity. Some vulnerabilities consist of multiple changes in different functions of the same file, which may affect the accuracy of different clone detection technique but not the dataset. Since in this study, we extracted every function, which was patched by the developers.

The quality of our collected vulnerability dataset fully depends upon the CVE [https://cve.mitre.org/], NVD [https://nvd.nist.gov/], and CVE-Details [https://www.cvedetails.com/] platforms. Since our vulnerability dataset was collected and verified from these sources. As we mentioned earlier that these sources are very trust-able as they have different evaluation criteria before they publish vulnerability.

Impact of key findings: The key findings from these vulnerability statistics directly impact the developer's behaviour. As it can be seen that vulnerabilities are increasing day by day, especially in OSS. Thus developer should be very careful during the development of different applications. As we can see in Fig. 11, the vulnerability *Execute Code* is leading vulnerability after *DoS*, which allows the hacker to execute his code without any authentication on the user system. So developer should be careful, by not using any vulnerable component, file, function or any code fragment, which has these security flaws. From this trend, we can see that Linux is leading in number of vulnerabilities as compare to other products because of open source product. Thus developer and reader can easily understand this trend, meanwhile hackers are exploiting this trend by injecting vulnerabilities directly into open source projects. Other key analysis helps a reader to examine the critical vulnerabilities, especially in OSS ecosystem. There are still a lot of vulnerable components which are being actively used today, including *Zeebuddy* and *Clipshare*. Thus, the developer or the user must always check the components to ensure that your component is the most recent one and the provider is providing the latest upgrades and updates of it.

8 Conclusion and future work

In this study, we proposed a technique to build a vulnerability benchmark at different levels of granularities including function-level, file-level and component-level, which helps us to overcome the cyber security attacks. In this study, we explored patch files of already published CVEs by tracing out the web links on different sources and locate the particular vulnerable source code. This vulnerable code always remain unfixed for many years and it always propagate to other systems gradually. Meanwhile, this study shows the recent cybercrime statistics, common hacking techniques, most often repair patterns in patch files, most important vulnerability types including zero-day vulnerabilities, vulnerability trends and some important ways to prevent from these crimes. This study is also have a learning point for software developers and hackers to understand these security standards very deeply. A sample vulnerability benchmark has been provided online, which can be used to trace vulnerabilities in your system. At component level granularity, the information of vulnerable components have also been provided, if you are using those components or a part of those components, you are vulnerable. For future concerns, we are planning to collect a big data-set of vulnerabilities and use it for the vulnerability detection in large-scale systems at large-scale level. Moreover, to build some vulnerability prediction models to detect vulnerable patterns.

9 Acknowledgment

This research was done in the key laboratory for information system security, Tsinghua University Beijing, Ministry of Education of China and it was supported by the National Natural Science Foundation of China.

10 References

[1] Avery, J., Wallrabenstein, J.R.: 'Formally modeling deceptive patches using a game-based approach', *Comput. Secur.*, 2018, **75**, pp. 182–190

[2] Kim, S., Woo, S., Lee, H., *et al.*: 'VUDDY: A scalable approach for vulnerable code clone discovery'. 2017 IEEE Symp. on Security and Privacy (SP), San Jose, CA, USA, 2017, pp. 595–614

[3] Jang, J., Agrawal, A., Brumley, D.: 'Redebug: finding unpatched code clones in entire OS distributions'. 2012 IEEE Symp. on Security and Privacy (SP), San Jose, CA, USA, 2012, pp. 48–62

[4] Brumley, D., Poosankam, P., Song, D., *et al.*: 'Automatic patch-based exploit generation is possible: techniques and implications'. IEEE Symp. on Security and Privacy, 2008 (SP 2008), Oakland, CA, USA, 2008, pp. 143–157

[5] Alhazmi, O.H., Malaiya, Y.K., Ray, I.: 'Measuring, analyzing and predicting security vulnerabilities in software systems', *Comput. Secur.*, 2007, **26**, (3), pp. 219–228

[6] Gorbenko, A., Romanovsky, A., Tarasyuk, O., *et al.*: 'Experience report: study of vulnerabilities of enterprise operating systems'. 2017 IEEE 28th Int. Symp. on Software Reliability Engineering (ISSRE), Toulouse, France, 2017, pp. 205–215

[7] Guo, H., Wang, Y.Y., Pan, Z.L., *et al.*: 'Research on detecting windows vulnerabilities based on security patch comparison'. 2016 Sixth Int. Conf. on Instrumentation & Measurement, Computer, Communication and Control (IMCCC), Harbin, People's Republic of China, 2016, pp. 366–369

[8] Charalampidou, S., Ampatzoglou, A., Chatzigeorgiou, A., *et al.*: 'Assessing code smell interest probability: a case study', Proceedings of the XP2017 Scientific Workshops, New York, NY, USA, 2017

[9] Svajlenko, J., Roy, C.K.: 'Fast and flexible large-scale clone detection with cloneworks'. Proc. 39th Int. Conf. on Software Engineering Companion, Buenos Aires, Argentina, 2017, pp. 27–30

[10] Akram, J., Shi, Z., Mumtaz, M., *et al.*: 'DroidCC: a scalable clone detection approach for android applications to detect similarity at source code level'. 2018 IEEE 42nd Annual Computer Software and Applications Conf. (COMPSAC), Tokyo, Japan, 2018, pp. 100–105

[11] Matsushita, T., Sasano, I.: 'Detecting code clones with gaps by function applications'. 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2017, New York, NY, USA, 2017

[12] Su, F.H., Bell, J., Harvey, K., *et al.*: 'Code relatives: detecting similarly behaving software'. Proc. 2016 24th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering, New York, NY, USA, 2016, pp. 702–714

[13] Al-Omari, F., Roy, C.K.: 'Is code cloning in games really different?'. Proc. 31st Annual ACM Symp. on Applied Computing, Pisa, Italy, 2016, pp. 1512–1519

[14] Keuning, H., Heeren, B., Jeuring, J.: 'Code quality issues in student programs', UU BETA ICS Departement Informatica, No. UU-CS-2017-006. ISSN 0924-3275, Aberdeen, UK, 2017

[15] Mondal, M., Roy, C.K., Schneider, K.A.: 'Identifying code clones having high possibilities of containing bugs'. Proc. 25th Int. Conf. on Program Comprehension, Buenos Aires, Argentina, 2017, pp. 99–109

[16] Hatano, T., Matsuo, A.: 'Removing code clones from industrial systems using compiler directives'. 2017 IEEE/ACM 25th Int. Conf. on Program Comprehension (ICPC), Buenos Aires, Argentina, 2017, pp. 336–345

[17] Krutz, D.E., Mirakhorl, M.: 'Architectural clones: toward tactical code reuse'. Proc. 31st Annual ACM Symp. on Applied Computing, New York, NY, USA, 2016, pp. 1480–1485

[18] Taibi, D., Janes, A., Lenarduzzi, V.: 'How developers perceive smells in source code: a replicated study', *Inf. Softw. Technol.*, 2017, **92**, pp. 223–235

[19] Tekchandani, R., Bhatia, R., Singh, M.: 'Code clone genealogy detection on e-health system using Hadoop', *Comput. Electr. Eng.*, 2017, **61**, pp. 15–30

[20] Abdalkareem, R., Shihab, E., Rilling, J.: 'On code reuse from StackOverflow: an exploratory study on android apps', *Inf. Softw. Technol.*, 2017, **88**, pp. 148–158

[21] Alam, S., Qu, Z., Riley, R., *et al.*: 'Droidnative: automating and optimizing detection of android native code malware variants', *Comput. Secur.*, 2017, **65**, pp. 230–246

[22] Akram, J., Shi, Z., Mumtaz, M., *et al.*: 'DCCD: an efficient and scalable distributed code clone detection technique for big code'. The 30th Int. Conf. on Software Engineering and Knowledge Engineering PROCEEDINGS SEKE 2018, San Francisco Bay, CA, USA, 2018, pp. 354–359

[23] Lerums, J.E., Poe, L.D., Dietz, J.E.: 'Simulation modeling cyber threats, risks, and prevention costs'. 2018 IEEE Int. Conf. on Electro/Information Technology (EIT), Rochester, MI, USA, 2018, pp. 0096–0101

[24] Srinivas, J., Das, A.K., Kumar, N.: 'Government regulations in cyber security: framework, standards and recommendations', *Future Gener. Comput. Syst.*, 2019, **92**, pp. 178–188

[25] Bolbot, V., Theotokatos, G., Bujorianu, M.L., *et al.*: 'Vulnerabilities and safety assurance methods in cyber-physical systems: a comprehensive review', *Reliab. Eng. Syst. Saf.*, 2019, **182**, pp. 179–193

[26] Smith, H., Morrison, H.: '*Ethical hacking: a comprehensive beginners guide to learn and master ethical hacking*' (CreateSpace Independent Publishing Platform, Scotts Valley, CA, USA, 2018)

[27] Prasad, Y.K., Reddy, D.V.S.: 'Review on phishing attack and ethical hacking', *Int. J. Res.*, 2019, **6**, (3), pp. 853–858

[28] Kumar, S., Agarwal, D.: 'Hacking attacks, methods, techniques and their protection measures', *Int. J. Adv. Res. Comput. Sci. Manage.*, 2018, **4**, (4), pp. 2353–2358

[29] Payne, B., Fagan, B.: 'Ethical hacking: teaching cyber safety from a hacker's point of view', 2019

[30] Akram, J., Liang, Q., Ping, L.: 'VCIPR: vulnerable code is identifiable when a patch is released (hacker's perspective)'. 2019 12th IEEE Conf. on Software Testing, Validation and Verification (ICST), Xian, People's Republic of China, 2019, pp. 402–413

[31] Chiew, K.L., Yong, K.S.C., Tan, C.L.: 'A survey of phishing attacks: their types, vectors and technical approaches', *Expert Syst. Appl.*, 2018, **106**, pp. 1–20

[32] Khatun, F., Islam, M.R.: 'Security in cloud computing-based mobile commerce'. Advances in Electronics, Communication and Computing, Rome, Italy, 2018, pp. 191–198

[33] Eling, M., Wirfs, J.: 'What are the actual costs of cyber risk events?', *Eur. J. Oper. Res.*, 2019, **272**, (3), pp. 1109–1119. Available at http://www.sciencedirect.com/science/article/pii/S037722171830626X

[34] Adams, S., Carter, B., Fleming, C., *et al.*: 'Selecting system specific cybersecurity attack patterns using topic modeling'. 2018 17th IEEE Int. Conf. on Trust, Security and Privacy in Computing and Communications/12th IEEE Int. Conf. on Big Data Science and Engineering (TrustCom/BigDataSE), New York, NY, USA, 2018, pp. 490–497

[35] Sevis, K.N., Seker, E.: 'Cyber warfare: terms, issues, laws and controversies'. 2016 Int. Conf. on Cyber Security and Protection of Digital Services (Cyber Security), London, UK, 2016, pp. 1–9

[36] Sheneamer, A., Kalita, J.: 'A survey of software clone detection techniques', *Int. J. Comput. Appl.*, 2016, **137**, pp. 1–21

[37] Liu, Z., Wei, Q., Cao, Y.: 'VFDETECT: a vulnerable code clone detection system based on vulnerability fingerprint'. 2017 IEEE 3rd Information Technology and Mechatronics Engineering Conf. (ITOEC), Chongqing, People's Republic of China, 2017, pp. 548–553

[38] Jiang, L., Misherghi, G., Su, Z., *et al.*: 'DECKARD: scalable and accurate tree-based detection of code clones'. Proc. 29th Int. Conf. on Software Engineering, Minneapolis, MN, USA, 2007, pp. 96–105

[39] Kim, D., Tao, Y., Kim, S., *et al.*: 'Where should we fix this bug? A two-phase recommendation model', *IEEE Trans. Softw. Eng.*, 2013, **39**, (11), pp. 1597–1610

[40] Nappa, A., Johnson, R., Bilge, L., *et al.*: 'The attack of the clones: a study of the impact of shared code on vulnerability patching'. 2015 IEEE Symp. on Security and Privacy (SP), San Jose, CA, USA, 2015, pp. 692–708

[41] Kim, S., Lee, H.: 'Software systems at risk: an empirical study of cloned vulnerabilities in practice', *Comput. Secur.*, 2018, **77**, pp. 720–736

[42] Li, Z., Zou, D., Xu, S., *et al.*: 'VulPecker: an automated vulnerability detection system based on code similarity analysis'. Proc. 32nd Annual Conf. on Computer Security Applications, Los Angeles, CA, USA, 2016, pp. 201–213

[43] Zhou, Y., Sharma, A.: 'Automated identification of security issues from commit messages and bug reports', Proceedings of 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17)], Paderborn, Germany, 2017, pp. 914–919

[44] Walden, J., Stuckman, J., Scandariato, R.: 'Predicting vulnerable components: software metrics vs text mining'. 2014 IEEE 25th Int. Symp. on Software Reliability Engineering, Naples, Italy, 2014, pp. 23–33

[45] Akram, J., Mumtaz, M., Gul, J., *et al.*: 'DroidMD: an efficient and scalable android malware detection approach at source code level', *Int. J. Inf. Comput. Secur.*, 2019, **11**, (1), p. 1

[46] Balzarotti, D., Monga, M., Sicari, S.: 'Assessing the risk of using vulnerable components'. Quality of Protection, Boston, MA, USA, 2006, pp. 65–77

[47] Pang, Y., Xue, X., Namin, A.S.: 'Predicting vulnerable software components through N-gram analysis and statistical feature selection'. 2015 IEEE 14th Int. Conf. on Machine Learning and Applications (ICMLA), Miami, FL, USA, 2015, pp. 543–548