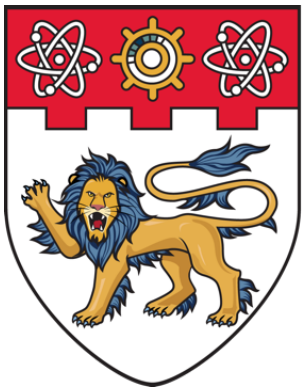


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Attack Surface Analysis and Code Coverage Improvement for Fuzzing

PENG LUNAN

School of Physical and Mathematical Sciences

2019

Attack Surface Analysis and Code Coverage Improvement for Fuzzing

PENG LUNAN

School of Physical and Mathematical Sciences

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Master of Science

2019

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research done by me except where otherwise stated in this thesis. The thesis work has not been submitted for a degree or professional qualification to any other university or institution. I declare that this thesis is written by myself and is free of plagiarism and of sufficient grammatical clarity to be examined. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

28/06/2019
Date

彭鲁楠
Peng Lunan

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it of sufficient grammatical clarity to be examined. To the best of my knowledge, the thesis is free of plagiarism and the research and writing are those of the candidate's except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

30 June 2019
Date


Wu Hongjun

Authorship Attribution Statement

This thesis **does not** contain any materials from papers published in peer-reviewed journals or from papers accepted at conferences in which I am listed as an author.

28/06/2019
Date

彭倫楠
Peng Lunan

Abstract

As cybercrime becoming a worldwide threat in the past decades, research on cybersecurity keeps attracting a great deal of attention. During a long time competition between attackers and defenders, vulnerability detection has been considered as the decisive pre-step for both sides. Among the massive methodologies of vulnerability detection, fuzzing test has demonstrated its outstanding performance on finding bugs automatically and effectively.

A fuzzer repeatedly provides generative-based or mutation-based samples to the target program to explore misbehavior of it. Even though many boosting techniques have been proposed to further improve the efficiency of fuzzing, nowadays there are still two crucial aspects remaining with enduring appeal to researchers: one is attack surface analysis to help fuzzers put more effort on the most potentially vulnerable locations, another one is code coverage improvement to guide fuzzers to explore more code regions.

In this thesis, we present attack surface analysis and code coverage improvement for fuzzing. In the first work, we choose Linux Kernel as the target, categorize its source files into different components upon their functionalities. Then we collect data of all related Common Vulnerabilities and Exposures (CVE) and analyze their distributive features to identify the vulnerable level of each component. In the second work, we utilize the rarely-hit edges as the metric to guide a multi-round generative-based fuzzing on Document Object Model (DOM) of Chromium browser. We use the default template to generate a large number of samples in the first fuzzing round, compute the hit times of all covered edges and find out samples that cover any rarely-hit edges as templates for the second round fuzzing. The approach achieved an obvious improvement on the code coverage of newly generated samples compared to the default one.

Acknowledgements

I would like to express my sincere thanks to everyone who has supported me in the past a couple of years. I do feel considerably grateful for all your help with both academic issues and daily life.

First of all, I would like to thank my supervisor, Professor Wu Hongjun, for his continuous and patient guidance throughout my whole study and work progress. His encouragement motivates me to keep exerting my full energy in solving any troubles and doubts faced during the work.

I would like to thank Professor Liu Yang for helping me with his expert insight and rich experience in the security research area. It is precisely because of his excellent guidance on my final year project during the undergraduate study that inspired my interest in cybersecurity research.

Furthermore, I would also like to thank my senior colleagues: Dr. Huang Tao, Dr. Wang Chenyu and Mr. Yu Haiwan, for providing significant advice and assistance to my research. They are all excellent friends to play with and good teachers to learn from.

Last but not least, I express my greatest gratitude to my parents. Their unconditional love equips me with unlimited determination and courage in my whole life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Cybersecurity

With the increasing reliance on computer and information technology, electronic crimes that target computers and networks, known as cybercrime, have caused unbelievable loss to society. In 2014, Reuters reported that the annual damage of cybercrime to the global economy is about $445 billion [2], including damage to business as well as to individuals. In 2017, WannaCry Ransomware Attack infected more than 230,000 computers in over 150 countries within one day, encrypted files on victim's computers without permission and asked for ransom money, even hit the UK's National Health service and caused unexpected emergencies [3]. Due to the growing threats from cybercrime, cybersecurity is attracting more and more worldwide attention.

Cybersecurity is a very wide field that concentrates on the protection of computer system and people's legal assets from electronic damage. The International Telecommunications Union (ITU) defined it as an assembly of policies, protocols, guidelines, tools, technologies, management approaches and assurance that can be used to ensure the safety of cyber environment [4].

Many countries have officially published statements and measures to fight against cybercrime such as system hacking and secrets stealing [5]. And companies with

business around cybersecurity are growing fast and strongly. Along with that, competition between nation and nation, company and company tends to be increasingly fierce. Therefore, there is no doubt that cybersecurity will continually play an important role in national safeguarding, global business, academic research and our daily life.

1.2 Vulnerability

The first necessary step required by hackers to launch a cybercrime is to find a vulnerability, which is defined as a weakness that allows an attacker to reduce a system's information assurance [6]. In this thesis, we only focus on software vulnerabilities rather than hardware or physical cases.

Software vulnerabilities exist mainly because of developers' carelessness and mishandling of abnormal conditions and will be triggered by specific inputs. Both attackers and developers try hard to discover these vulnerabilities. The attackers aim to build malformed inputs accordingly to break protections of the target. While the developers need to locate and fix these vulnerabilities for enhancing safeguarding for their product.

To collect and summarize as many software vulnerabilities as possible, the Common Vulnerabilities and Exposures (CVE) system was developed. Vulnerabilities in publicly released and widely used software packages and datasets can be submitted to this system. After verification and evaluation, unique CVE Identifiers, as well as Common Vulnerability Scoring System (CVSS) scores, will be assigned based on their impactive level on confidentiality, integrity and availability of affected target.

The number of assigned CVEs provided by CVE system [7], shown in Figure 1.1, has indicated an obviously increasing trend in the past decades. By comparing CVSS score distribution in recently three years in Figure 1.2 [1], it can be figured out that the amount of CVEs with a score lower than 8 is in explosive growth, indicating the field of vulnerability discovery and submission is attracting a mass of individuals and organizations.

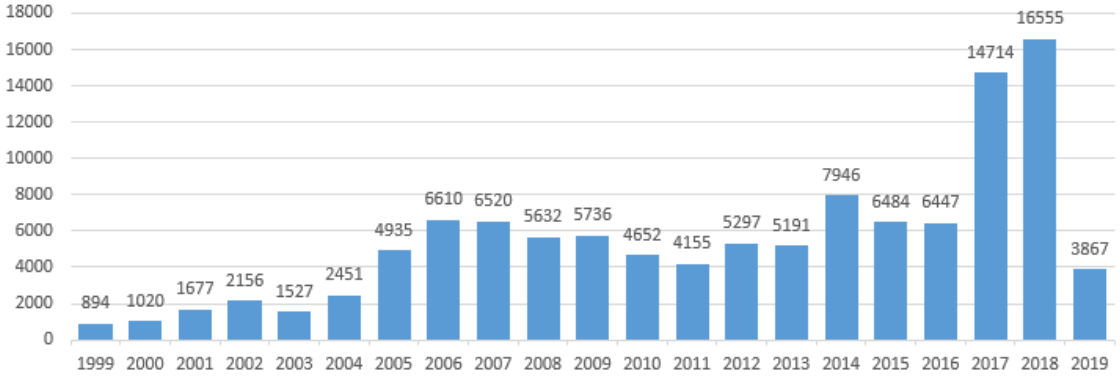


FIGURE 1.1: Number of CVEs (1999 Jan - 2019 May)

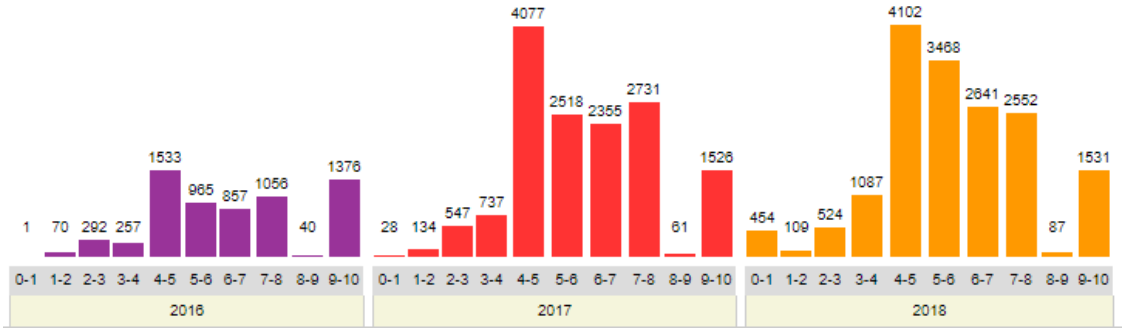


FIGURE 1.2: CVSS Score Distribution [1]

1.2.1 Types

There is no authoritative standard of classification for vulnerabilities in software due to the variety of root causes. In this thesis, we introduce four common types of software vulnerabilities.

- **Overflow**: Includes but not limited to Integer Overflow and Buffer Overflow.

  Integer Overflow occurs when assigning a numeric value that is out of representable range to a certain type of variable. For example, assign the result of 255+2 to an 8-bit unsigned variable, whose value range is from 0 to 255, will trigger an integer overflow, causing the unsigned variable to be 1 instead of 257. It may cause security bugs in case the overflowed value is used to define some crucial number, such as the size of memory allocation.

  Buffer Overflow occurs when program overruns the boundary of a buffer and overwrites adjacent memory address. For example, if we define A as a string buffer with a size of 5 bytes, without bounds checking, assigning

an 8-byte string to A will cause buffer overflow and overwrite the next 3 bytes. Exploitation on buffer overflow varies by the memory architecture. Usually, there are stack-based and heap-based. Figure 1.3 demos a simple stack-based buffer overflow exploitation, which overwrites the return address of the vulnerable stack, results in arbitrary address jumping.

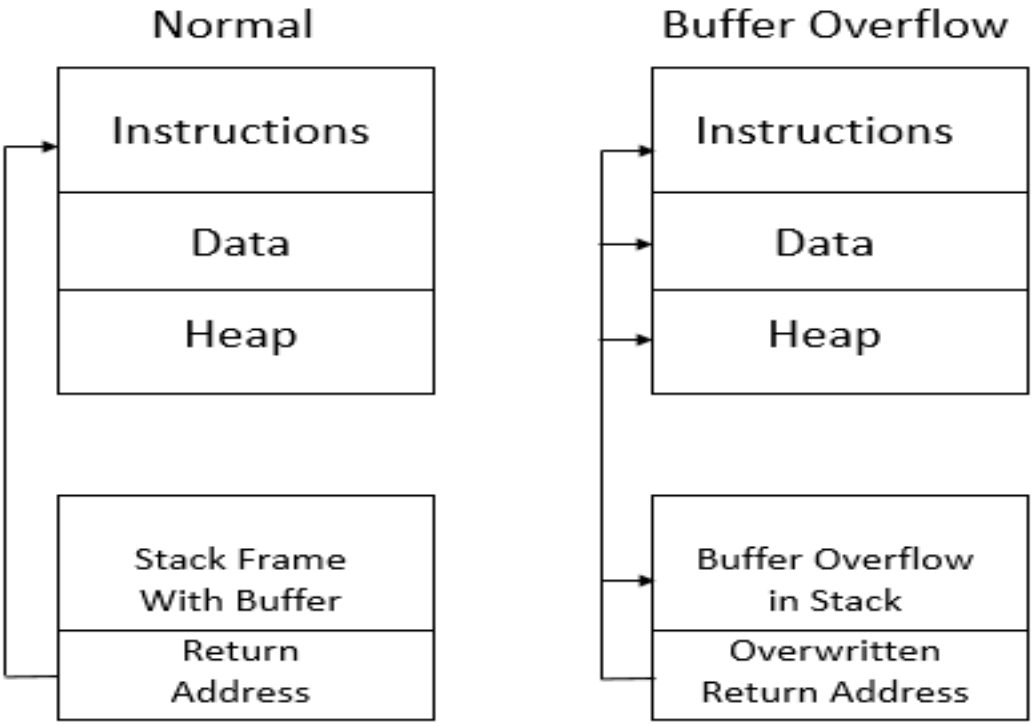


FIGURE 1.3: Stack-based Buffer Overflow Exploitation

- **Use After Free**: Refers to the abnormal attempts to access memory address which has already been freed. Usually caused and exploited by a dangling pointer.

  In computer programming, a pointer points to a specified memory address that is allocated for storing values. Programmer can obtain these stored values by referencing the correlative pointer. That significantly reduces the overload of repetitive operations such as string traversal and tree traversal. If a pointer is not properly handled after freeing its pointed memory, it becomes a dangling pointer, also called a wide pointer, and consequently may lead to a memory corruption flaw. Attackers detect and make use of dangling pointers to violate memory and launch an exploitation.

One example of use after free is shown in Figure 1.4. Once the memory address is freed, the pointer referencing to it becomes a dangling pointer. If the attacker is able to write malicious code to this freed memory and the dangling pointer is reused after that, the malicious code may get executed, and the pointer turns to be an accomplice for attacker's evil purpose.

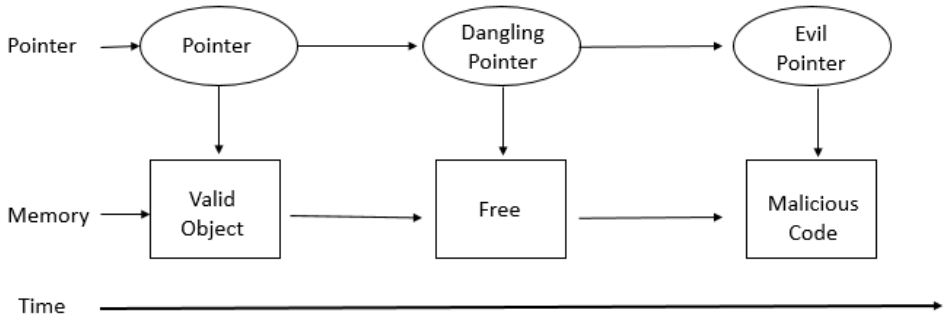


FIGURE 1.4: Use After Free

- **Race Condictions**: Occurs when the timing or sequence of processes execution gets into confusion.

  Modern software and application usually take advantage of multithreading to achieve better performance. In the case that different threads share some same states or have high dependence between each other, the execution sequence must be strict and exclusive. Otherwise, the possibility of corrupting will raise, resulting in undefined behavior.

  For example, there are two threads both read an variable A and increase it by one, then write back. The normal execution sequence is first running *thread 1*, after completion, run *thread 2*, finally get $A=2$ (initially $A=0$). However, in a race condition, the sequence may be changed to that in Figure 1.5, *thread 2* reads variable A before *thread 1* writes the new vaule back. The final value of A hence becomes 1 instead of 2.

- **Input Validation**: Arises when the improper user input is unchecked.

  User interaction is a significant aspect of software development. Many software is designed to take and handle inputs from the user and provide output to solve their inquiries or requests. If an input is not suitable for the requirement, such as inputting a string when an integer is expected, and without

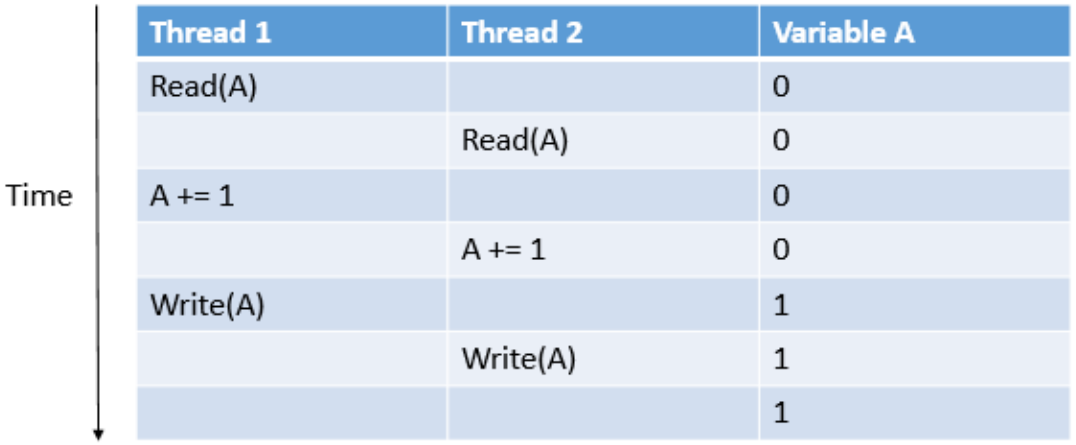


FIGURE 1.5: Race Condition

properly checking, error may occur. In some worse situations, it can be used for security exploitation.

One exploiting technique based on that is Structured Query Language (SQL) Injection. SQL is a widely used programming language for processing and managing data in a database [8]. In a SQL-based application with a lack of input check, malicious SQL statements can be inserted and get executed. For example, the following statement intends to query records from table of users with specified username in background database:

SELECT * FROM *users* WHERE *name* = *(INPUT)*;

Attacker can obtain the records of all users rather than the inputted one, by simply inject input as a short statement, like the following:

SELECT * FROM *users* WHERE *name* = '' OR '1' = '1';

In that case, since '1' = '1' is always TRUE, all records in the table will be selected, causing information leakage.

1.2.2 Detection

Since vulnerability is so crucial in defense of cybercrime, massive research about its detection has been conducted, and many detection tools have been developed. There exist multiple approaches to detect software vulnerabilities. According to

the methodology, we can categorize these approaches into two basic types: static approach and dynamic approach.

- **Static Approach**: Analyze and examine the source code of the target application without executing it.

  Static analysis aims to extract information such as data chunk, integrity constraints, data flow and control flow, from source code and launch a judgment based on that. Sotirov introduced that the most popular approaches to static analysis include but not limited to pattern matching, abstract syntax tree (AST) analysis and taint checking [9].

  Pattern matching is to firstly summarize sequence patterns of specific abnormality, then check the matching condition to detect it in the target program. AST represents the abstract syntax of target source code in a tree structure, within that every construct is parsed into a node, and can be used to analyze both syntax and semantic rules of the program. Taint checking proposes to proceeds variables one by one and obtains a set of variables with a potential probability of being influenced by outside inputs, therefore the checker is able to alert users when an influenced variable is used in some critical statements.

  The above approaches are widely used in software vulnerability detection. Van Lunteren [10] and Dharmapurikar [11] build pattern-matching models for intrusion detection. Skyfire applies AST [12] analysis to implement a seed generator for software testing. And Leakminer [13] is designed to detect information leakage vulnerability on Android by static taint analysis.

  The biggest benefit of the static approach in vulnerability detection is the outstanding capacity of detecting bugs deeply hidden in rarely reached code blocks. But it also has many obvious limitations. Firstly, static analysis is not suitable for closed-source applications. Secondly, approaches like pattern matching highly depend on the quality of summarized rules and patterns, that requires deep knowledge and rich experience in the related area.

- **Dynamic Approach**: Analyze the software and detect abnormities based on the performance during execution.

In plain words, it is to execute the target application with sufficient valid (both syntactically-valid and semantically-valid) inputs, monitor the process, and dump pre-defined interesting behavior for tracking and analyzing, aiming to find out the root cause and identify potential vulnerabilities.

Compared to static analysis, because of black-box testing technique, open-source is not required for detecting vulnerabilities dynamically. Dynamic testing is also under a low-level requirement of knowledge on the target program. However, multiple assistant means, like test case minimization and code coverage, are needed to improve the efficiency and accuracy of dynamic detection [14, 15]. In some cases, symbolic execution and loop analysis are also required for overcoming complicated magic number and dead loop issues to trigger deeper paths [14, 16–18].

In this thesis, we focus on fuzzing, one of the most commonly used dynamic approach for detecting software vulnerabilities. More details will be introduced in the next subsection.

1.3 Fuzzing

Fuzzing is firstly publicly mentioned in a research project of the University of Wisconsin in 1988 [19]. Sutton, Greene and Amini define fuzzing as *a method for discovering faults in software by providing unexpected inputs and monitoring for exceptions* [20]. It is an automated technique for software testing, and has been proved as one of the most effective testing methodologies by the fact that a huge amount of software vulnerabilities have been detected based on it in the past decades. Because of its high efficiency, famous vendors such as Microsoft [21] and Google [22] keep spending more and more resources on developing fuzzers in recent years.

1.3.1 Basic Approach

The simplest approach of a fuzzing framework is shown in Figure 1.6. It consists of only three components: inputs, target program and execution monitor. The framework collects sufficient inputs (also known as test cases) through automatic generation, self-mutation or crawling from the Internet. Then it repeatedly sends these inputs to the target program to execute. A monitor is attached during the execution, to discover and dump any exceptions. Valuable exceptions will be analyzed manually by using disassembler or debugger, such as Interactive Disassembler (IDA) [23] and GNU Debugger (GDB) [24], to be further classified as vulnerable or vulnerable-free.

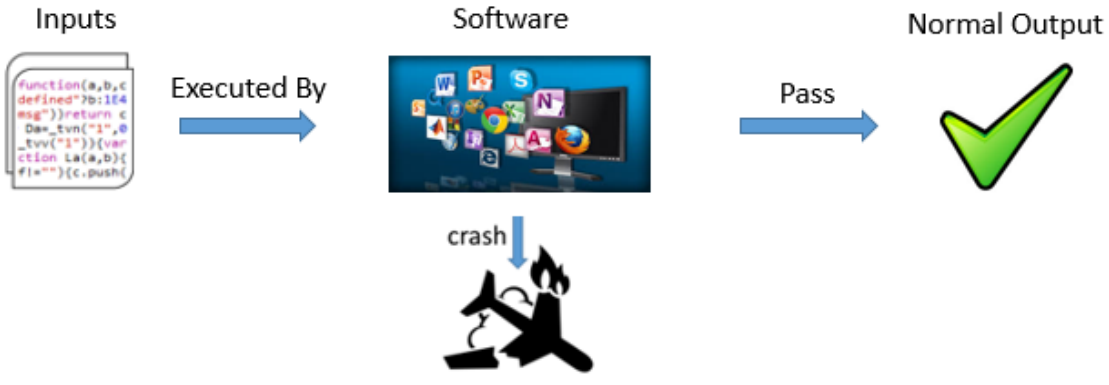


FIGURE 1.6: Basic Approach of A Fuzzing Framework

A fuzzer can be categorized by different policies. Depending on the methodology of test case collection, it can be categorized into generation-based fuzzing or mutation-based fuzzing. Depending on the awareness of the target program's structure and rules, it can be categorized into white-box, black-box or grey-box fuzzing.

- **Generation-based**: Be aware of syntax structure and semantic rules of input, automatically generate new valid inputs accordingly. Fuzzing frameworks such as Domato [25] and LangFuzz [26] that focus on the input generation approach are all generation-based.
- **Mutation-based**: Launch mutative strategies such as bit flipping and byte flipping on existing inputs. Mutating these provided inputs to generate new

inputs. TaintScope [27], Driller [17] and American Fuzzy Lop (AFL) [28] belong to this category.

- **White-Box Fuzzing**: Fuzzing on programs whose source code is available. Since the internal structure is visible, information such as control flow, data flow and code coverage are relatively simple to get. Therefore, a white-box fuzzing framework like SAGE [29] usually leverages static analysis as well as symbolic execution to perform fuzzing.

- **Black-Box Fuzzing**: Opposed to white-box, uses a massive amount of inputs to fuzz target program in the condition that internal structure is unaware. Most black-box fuzzing frameworks such as KameleonFuzz [30] and PULSAR [31] make use of high-quality vulnerable patterns summarized by themselves, combining with output-based learning, to develop their fuzzing strategy.

- **Grey-Box Fuzzing**: Partially aware of internal structure, leverages lightweight instrumentation to fetch transition and coverage information, in order to integrate with black-box fuzzing's methodologies and resulting in a reasonable balance between accuracy and execution speed. Some famous fuzzing framework such as LibFuzzer [32] and AFL [28] are based on grey-box approaches.

1.3.2 Guided Approach

One extreme efficient extension to fuzzing is guiding the approach by some feedback, which is known as guided fuzzing.

In a guided approach of fuzzing, some aspects of outcome produced by previous inputs are recorded. Through specified treatment, inputs with valuable feedback will be kept and marked as "interesting", while "uninteresting" inputs are discarded. These interesting inputs can be viewed as high-quality seeds and will be used as bases of mutation or templates of generation for creating new inputs. The simple algorithm of guided fuzzing approach is shown in Algorithm 1.

Algorithm 1 Guided Fuzzing Algorithm

Input: Inputs(Test cases) in a queue: Q; Generation/Mutation approach: M; Feedback judgement approach: F; Target program: T

```
1: while Q is not empty do
2:     S ← Dequeue(Q)
3:     for Pre-defined times of Generation/Mutation on S do
4:         S' ← M(S)
5:         R ← Execute(S', T)
6:         if Crash occurs in R then
7:             Save R and S';
8:         else
9:             if F(R) is interesting then
10:                Update F based on R;
11:                Enqueue(Q, S');
12:            end if
13:        end if
14:    end for
15: end while
```

From Algorithm 1, we can see the crucial aspect of the guided fuzzing approach is how to judge whether an executive result is interesting, in other words, the metric of judgment. That has a decisive influence on the efficiency of fuzzing.

In Chapter 3 of this thesis, we use *code coverage*, one broadly used metric in recent fuzzing research like [14, 15, 28, 33, 34], to launch a coverage-guided approach for fuzzing JavaScript engines and browsers. More background will be discussed in that chapter.

1.4 Thesis Organization

In this thesis, we will present two works related to software vulnerability detection. The first one is attack surface analysis for fuzzing, and the second one is code coverage improvement for guided generative fuzzing on DOM of Chromium browser.

The rest of this thesis is organized as follows:

Chapter 2 presents the work about attack surface analysis on Linux Kernel by collecting related vulnerabilities and mapping them to kernel components. We first introduce some background of the attack surface and Linux Kernel fuzzing. Then we present the flow of our vulnerability collection and mapping work. Finally, identify the vulnerable level of each component by analyzing the distributive status of these vulnerabilities.

Chapter 3 presents the work about improving code coverage of generative fuzzing on Chromium DOM through a rarely-hit edges targeted strategy. We will first describe the background of coverage-guided fuzzing. Then explain in detail the steps of our approach. Finally, we will present our implementation, discuss experiment outcomes, and evaluate its performance on coverage improvement.

Chapter 4 summarizes our works and discusses what can be further improved in the future.

Chapter 2

Attack Surface Analysis on Linux Kernel

2.1 Background

2.1.1 Attack Surface Analysis

The attack surface of a system is defined as *the set of ways in which an adversary can enter the system and potentially cause damage* [35]. In more detail, it is the sum of all valuable data, all paths lead to these data and all code protects these paths and data in a system [36]. Larger attack surface usually indicates weaker security.

Attack surface analysis focuses on proceeding a security evaluation for target system or application, aims to obtain a vulnerability assessment, figure out potential security risks, identify vulnerable parts and finally assist developers to fix these vulnerabilities and enhance the safeguard. It is important for both developers and attackers to find and understand vulnerable areas in the target, because of a brief overview of which parts are at high risk can be summarized and then benefits the vulnerability detection.

Since it can predict to some extent where vulnerabilities may exist, attack surface analysis is usually viewed as powerful assistance for efficient vulnerability detection. Here we discuss some related work about attack surface analysis, as well as its assistant application in vulnerability prediction and detection:

Howard, Pincus and Wing [37] propose a multi-dimension metric for measuring the security of a system. They summarize the used resources, communication channels & protocols, and access rights of a given attack as three dimensions for describing a system's attack surface, and use a count of attack opportunities to indicate a system's "attackability", which is treated as a measurement of how exposed the attack surface is.

Shin and Williams [38] make use of code complexity metrics to predict security vulnerabilities. According to the correlation between software vulnerability and complexity, they extract nine code complexity metrics to categorize functions into vulnerable, none-vulnerable and faulty functions. They also develop a predictor based on that and demonstrate its capacity of distinguishing vulnerable functions.

LEOPARD [39] is a framework designed to assist the identification of potential vulnerability functions. Complexity metrics and vulnerability metrics are combined in this framework. The former one is used to group functions, and the latter one performs as a rank standard to rank these functions in descending order of vulnerable level. The application of LEOPARD on fuzzing of real software indicates an outstanding performance.

2.1.2 Linux Kernel Fuzzing

To ensure the safety of memory and hardware, the virtual memory of an operating system is usually segregated into kernel space and user space. The core in an operating system is running in the kernel space, known as system kernel, which is protected by the highest privilege and takes full control over the whole system.

The kernel is responsible for directly controlling the computer hardware such as Central Processing Unit (CPU), Random-access Memory (RAM) and storage device. Meanwhile, it interfaces with user space processes through system calls. The brief architecture is shown as Figure 2.1.

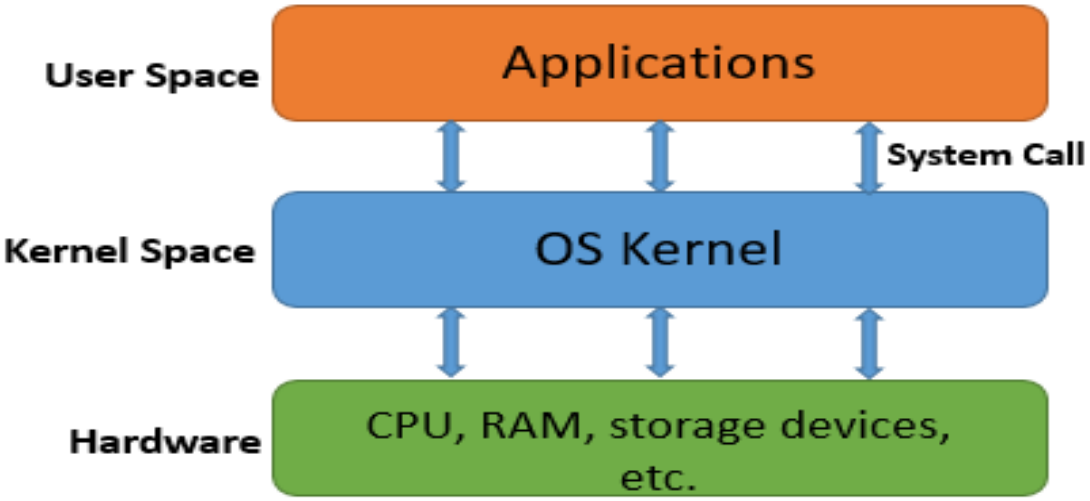


FIGURE 2.1: User Space and Kernel Space

Linux Kernel is an open-source kernel for the Linux family of operating systems. It was created by Linus Torvalds in 1991 [40], written mainly in C language and compiled by GNU Compiler Collection (GCC). As of November 2017, all of the top 500 supercomputers in the world use Linux as the system kernel [41]. In Linux Kernel, partial code such as device drivers is treated as loadable kernel modules (LKMs) and can be loaded once required and unloaded to free memory after usage.

Since kernel holds the root privilege for accessing everything in a computer system, successful exploitation on kernel usually results in complete control of the system. Because of that, kernel vulnerability detection keeps attracting the attention of participators in cybersecurity and cybercrime. In recent years, as fuzzing demonstrates its power in vulnerability detection, a lot of research around kernel fuzzing is published. Here we discuss some related work that aims to detect vulnerabilities in OS kernel by fuzzing:

Trinity [42] is a fuzzing framework designed to detect Linux Kernel vulnerability through testing Linux system calls. Instead of calling system calls with random arguments, Trinity builds a bunch of sockets for a certain type of argument, such

as file descriptor parameters. Once such an argument is required for a system call, the fuzzer will select one randomly from the bunch, aiming to avoid the problem that the kernel will reject a pure random invalid parameter by returning -EINVAL.

DIFUZE [43] focuses on fuzzing kernel drivers, which are essential for the kernel to interface with physical devices, ranging from data storage devices to camera, speaker and sound card. It utilizes static analysis to generate valid inputs of target drivers in an analysis host, then run these inputs to trigger the corresponding operations in kernel drivers in an external execution host. The input sequence is logged and execution results are transferred back, for manual analysis when the target host runs into a crash.

kAFL [44] is a hardware-assisted fuzzing framework for operating system kernel. It makes use of *Interl Processor Trace* to obtain information about process execution and traces of branches, then leverages the information as feedback to guide the further fuzzing approach. kAFL also segregates its framework into two components: a hypervisor to handle fuzzing logic, produce branch coverage and generate new test cases accordingly, and a target virtual machine to execute the test cases. The two components communicate with each other via hypercalls, in order to implement data transfer and minimize the expensive execution overhead such as a reboot.

2.2 Motivation and Approach

One of the most acknowledged challenging problems in Linux Kernel fuzzing is its extreme high complexity. Until September of 2018, the repository of Linux Kernel source tree contains more than 61 thousand files, 782 thousand commits (a set of changes in a repository) and 25 million lines of code [45]. This level of complexity leads to an unclear attack surface, so the fuzzer's designer is hard to identify the most vulnerable part of Linux Kernel.

The main motivation of our work in this chapter is to find out the top vulnerable parts or components of Linux Kernel, through that guiding the fuzzing framework to avoid wasting time on well-protected parts and put more energy on these vulnerable parts, to achieve a higher fuzzing efficiency for Linux Kernel.

We perform the analysis as follows:

1. Collect existing Linux Kernel CVEs in past decades.
2. Search and crawl all commits that are proposed to patch Linux Kernel CVEs, identify the files infected by each of these commits.
3. Categorize Linux Kernel components, figure out what files are included in each component.
4. Summarize and analyze the collection results. Map collected CVEs to categorized components. Figure out frequent vulnerability types, vulnerable components and vulnerable Linux Kernel source files.

2.3 Crawler Design

In this work, we develop a web crawler to crawl needed data from the Internet.

The crawler is written in Python code and mainly based on three Python libraries: `threading`, `requests` and `BeautifulSoup` [46]. Its work flow is shown in Figure 2.2

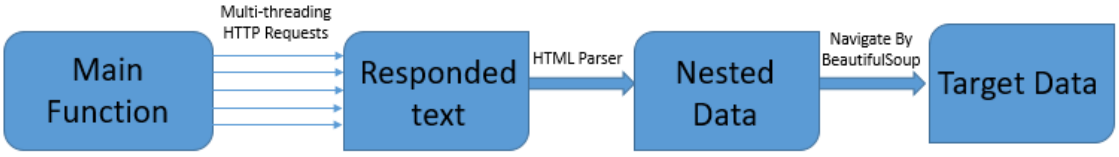


FIGURE 2.2: Workflow of Our Crawler

Firstly, we identify the target URLs in the main function and create multiple threads. Each thread will proceed a function which takes a specified URL as input to send an HTTP request for the given URL, and get the responded HTML text through the command `html=requests.get(url).text`.

Secondly, we utilize `BeautifulSoup`, a Python library used for pulling data out of HTML files, together with lxml's HTML parser [47] to parse the responded html text and obtain nested html data. The command is `soup=BeautifulSoup(html, 'lxml')`. This nested data contains complete html code of the given URL.

Finally, we navigate that nested data by using methods provide by `BeautifulSoup` to extract data we need. For example, the `find_all()` mtehod will extract all tags in the html data that match filters defined by us, the filter can be tag name, attributes and contained text. One instance is the command `soup.find_all('div', attrs='class' : "A", 'id' : "B")` which returns a list of all tags whose `name='div', class='A'` and `id='B'`.

In summary, the workflow of our crawler is very straightforward. The crucial part is how to find data source URL and how to extract specified data accurately. The former needs manual search and decision, and the latter highly depends on the HTML code structure of the chosen source.

2.4 CVE Collection

We propose to collect the existing publicly known vulnerabilities of Linux Kernel that are assigned unique CVE IDs. The significant information of a CVE vulnerability we want to get includes published year, ID, CVSS score, types and its patch commits.

2.4.1 Database Choice

There are two famous databases of CVE security vulnerabilities. One is National Vulnerability Database (NVD) [48], a repository managed by the U.S. government. The other one is CVEdetails [49], a free CVE vulnerability information source. Both of them provide information about a specified vulnerability such as its assigned ID, CVSS score, description, impact details, types and links for references.

In this work, we prefer to collect Linux Kernel related CVEs from CVEDetails. The main reason is, although both CVEDetails and NVD provide the list of affected products for every CVE vulnerability which enables us to get a statistic list for CVEs that affect Linux Kernel, searching results in NVD contain many unexpected false positives.

One false positive instance of searching in NVD is `CVE-2018-1857` [50], which is an information leakage vulnerability found in IBM DB2, a family of data management products. Through viewing its advisory, we can find that the root cause is located in DB2 itself, but not in Linux Kernel.

This false positive occurs because NVD includes Linux Kernel as one of the running platforms for DB2, as shown in Figure 2.3, and wrongly put this CVE in the return list when we search vulnerabilities by specifying Linux Kernel included in the affected products. We can further find many more similar false positive results in this style in NVD, such as `CVE-2018-1786` [51] and `CVE-2018-1834` [52].

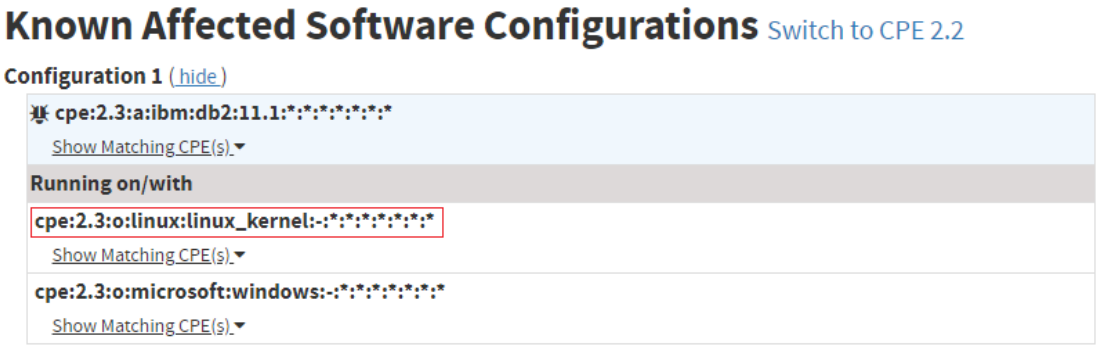


FIGURE 2.3: Known Affected Software Configurations of CVE-2018-1857

Instead of crawling from NVD, we can get a CVE list related to Linux Kernel from a vulnerability statistics webpage [53] provided by CVEDetails. These CVEs are divided by published year, and they all have Linux Kernel included in their affected products. So there is no false positive.

Finally, we summarize significant indicators about Linux Kernel CVE collection in the above two databases and show in Table 2.1. These two databases both contain all needed information of a vulnerability for us, such as published year, ID, score, types and commit links. Through searching CVEs published from 1999 to 2018 as Linux Kernel included in affected products, NVD reports 3543 results with many false positives. CVEDetails reports less as 2162, but the advantage is free of false positive. Since false positive results will surely have very disruptive impact on our analysis, we chose CVEDetails as the source of CVE info collection.

Database	Significant Info	Number of Results (1999 - 2018)	False Positive
NVD	All Contained	3543	Many
CVEDetails	All Contained	2162	None

TABLE 2.1: Database Comparison in aspect of Linux Kernel CVE Collection

2.4.2 Collect ID, Type, Score

Basic information of a CVE vulnerability we need includes its ID, type and CVSS score.

CVEDetails divides CVEs that affect Linux Kernel into multiple lists according to published years. And the list of each year may be shown in multiple webpages since CVEDetails lists only 50 CVEs at most on one webpage. In this case, we must first find out all the webpage links that reference to CVE lists we want.

The basic URL of Linux Kernel CVEs in CVEDetails is `https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/`, where `vendor_id-33` specifies the affected vendor is Linux and `product_id-47` specifies the affected vendor is Linux Kernel. To obtain the list of a specified published year, we only need to append `year-$Y` with `$Y` specifies the years behind it. For example, the following URL will direct us to a webpage which lists Linux Kernel CVEs published in the year 2005:

`https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2005`

By visiting above URL, we find the number of Linux Kernel CVEs publish in 2005 is 133, and these CVEs are listed in 3 webpages referenced by 3 hyperlinks. In HTML text of above URL, the hyperlinks are stored in a `<div>` tag with `class="paging"` and `id="pagingb"`, and this `<div>` tag is unique, as shown in Figure 2.4. So we can obtain this tag through command `page = soup.find('div', attrs='class' : "paging", 'id' : "pagingb")`, where `soup` stores the nested html data after parsing. Nextly, use command `page.find_all('a', href=True)` to obtain all the

<a> tags, whose element in name of 'href' can provide us the hyperlink that references to a webpage containing our needed CVE lists.

```
<div class="paging" id="pagingb">
    Total number of vulnerabilities : <b>133</b>  
Page :
<a  href="/vulnerability-list.php?
vendor_id=33&product_id=47&version_id=&page=1&hasexp=0&opdos=0&opec=0&opov=0&opcsrf=0&opgpi
ad925965de7b8e459c929b2bf1f"     title="Go to page 1">1</a>
(This Page)<a   href="/vulnerability-list.php?
vendor_id=33&product_id=47&version_id=&page=2&hasexp=0&opdos=0&opec=0&opov=0&opcsrf=0&opgpi
ad925965de7b8e459c929b2bf1f"     title="Go to page 2">2</a>
<a  href="/vulnerability-list.php?
vendor_id=33&product_id=47&version_id=&page=3&hasexp=0&opdos=0&opec=0&opov=0&opcsrf=0&opgpi
ad925965de7b8e459c929b2bf1f"     title="Go to page 3">3</a>
</div>
```

FIGURE 2.4: tag contains hyperlinks to CVE lists

We collect 55 hyperlinks for CVE lists from 1999 to 2018. Next step is to collect CVE ID list for each year. We send request to these 55 hyperlinks in multiple threads, and get responded html. After parsing and look through the html text, we find that every CVE ID is stored in an <a> tag like follows:

```
<a href="/cve/CVE-2005-4811/" title="CVE-2005-4811 security
vulnerability details">CVE-2005-4811</a>
```

All these <a> tags can be fetched through command `CVE_List = soup.find_all('a', href=True,string=re.compile("^CVE-"))`, here `string=re.compile("^CVE-")` is used to match tags whose string starts with "CVE-". Then CVE ID like above, `CVE-2005-4811`, is extracted by query `'.text'` of elements in `CVE_List`.

We collect IDs of all the 2162 Linux Kernel CVEs provided by CVEDetails. Next we need to collect types, scores and reference links of them.

The details of a CVE can be viewed in **https://www.cvedetails.com/cve/$ID**, where `$ID` specifies its ID. We multi-threadedly request URLs for all the 2162 CVE IDs, then extract required data through the following methods:

- **Types**:

  Types of a CVE are stored in a table row whose header is `Vulnerability Type(s)`. So we use command `Type = soup.find('th', text="Vulnerability`

`Type(s)")` to locate target table row, then use `Type = Type.find_next('td')` and `Type_List = Type.find_all('span')` to extract all the types stored in `<span>` tags in the table cell. For example html text segment in Listing 2.1, through above commands we can obtain a list of two strings inside: `['Overflow','Gain privileges']`, which are the types categorized by CVEDetails.

```
<tr>
<th>Vulnerability Type(s)</th>
<td>
<span class="vt_overflow">Overflow</span>
<span class="vt_priv">Gain privileges</span>
</td>
</tr>
```

LISTING 2.1: Example `<tr>` tag that contains CVE types

- **CVSS Score**:

  CVSS Score is stored in the same table as types, but the `<span>` tag containing it has a specific class name: `"cvssbox"`. So it can be obtained directly by command `soup.find('div','class' : "cvssbox").text`.

2.4.3 Collect Patch Commits

CVEDetails usually provides multiple reference links for a CVE. These links point to messages, discussion, advisories or patches relevant to fixing this CVE. Figure 2.5 show an example of how the reference links table looks like in CVEDetails.

– References For CVE-2017-9059

http://www.securityfocus.com/bid/98551
BID 98551

https://bugzilla.redhat.com/show_bug.cgi?id=1451386 CONFIRM

https://github.com/torvalds/linux/commit/c70422f760c120480fee4de6c38804c72aa26bc1 CONFIRM

https://www.spinics.net/lists/linux-nfs/msg63334.html CONFIRM

http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=c70422f760c120480fee4de6c38804c72aa26bc1 CONFIRM

FIGURE 2.5: An Example of Reference Links Provided by CVEDetails

All the reference links of a CVE are stored in a table with `id=id="vulnrefstable"` in its html text. Each link takes one cell, means one `<td>`, to store, and has a `<a>` tag with attribute `target="_blank"` to create the link or hyperlink.

In this work, we are only interested in links that direct us to the commits that update Linux Kernel source code to put a patch. Through research, we find an online repository [54] that provides the details of Linux Kernel commits. User can request the detailed information of a commit through the following URL with a specified `$id`:

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=$id

So the target reference links we want should be in the above URL format. We look at Figure 2.5 again and observe that the last link in red rectangle suits this format (although in the middle it is `cgit/` instead of `pub/scm/`, this link still directs us to where we want). Therefore, we use the command `soup.find_all('a','target' : "_blank",text=re.compile("git.kernel.org"))` to fetch all `<a>` tags that have `target="_blank"` and include `"git.kernel.org"` in the text, then extract their text to get URLs link to patch commits of the CVE.

Once we get URLs of patch commits, we can automatically access the their detailed information. These information includes its author, date, hash id, parent commit, download link and change logs. The most significant part for us is the changelogs, which are stored in a .diff file and used to record addition and deletion of code segments, as well as the names of modified files and functions, just like what Figure 2.6 shows.

In this work, we only care about the names of the modified files that can be found in `<a>` tags nested in a `<td>` tag with `class="upd"`. We extract all the target `<td>` tags through command `soup.find_all('td', 'class' : "upd")`, then for each element in the returned list, use `.find_next('a',href=True).text` to obtain the file name.

It should be noted that some CVEs have no patch commit while some may have multiple. After crawling, we find that 1189 of total 2162 CVEs have at least one patch commit, and the commits of 62 of them are no longer valid. For the 1127

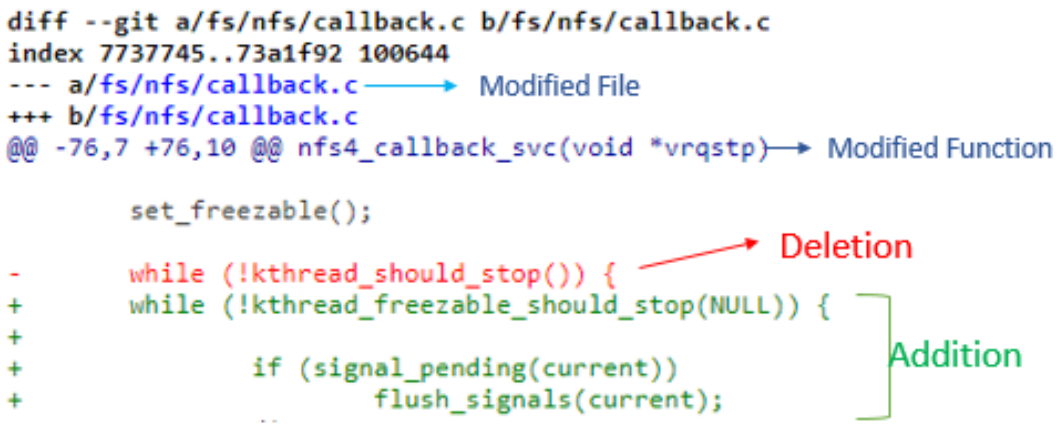


FIGURE 2.6: An Example of commits change log

CVEs with valid commits, we collected 1310 commits and in total 2492 files are modified in these commits.

2.5 Linux Kernel Component

We need to categorize Linux Kernel components and collect all file names that are included in each component for further CVE-Component mapping and analysis.

2.5.1 Component Category

Currently there is no restrictive official standard for categorizing Linux Kernel components. People usually categorize Linux Kernel code into different components upon their functionalities. For example, Jones [55] decomposes it into 7 major components: **system call interface**, **process management**, **memory management**, **virtual file system**, **network stack**, **device drivers** and **architecture dependent code**.

For attack surface analysis, categorizing in more details leads to a better result. In this work, we reference a common functionality-based but more detailed category method proposed by Constantine in his Linux Kernel Map [56]. This method first follows functionality to classify 6 major components : `human interface`,

`system`, `processing`, `memory`, `storage` and `networking`. Then the major components are further decomposed into 37 components based on 6 different layers: `user space interface`, `virtual`, `bridges`, `logicalm device control` and `hardware interface`.

The table of components is shown in Figure 2.7. The horizontal headers are functionalities and vertical headers are layers. There are 37 components and some of them crosses multiple functionalities or layers.

Func / Layers	Human interface	system	processing	memory	storage	Networking
user space interface	HI char device	Interfaces core	Processes	Memory access	Files & directories access	Sockets access
Virtual	Security	Device Model	Threads	Virtual memory	Virtual file system	Protocol families
Bridges	Debugging		Synchronization	Memory mapping	Page cache / Swap	Networking storage / Socket splice
Logical	HI subsystems	System run	scheduler	Logical memory	Logical file systems	protocols
Device control	Abstract devices and HID class drivers	Generic HW access	Interrupts core	Page allocator	Block devices	Network interface
Hardware interfaces	HI peripherals device drivers	Device access and bus drivers	CPU specific	Physical memory operations	Disk controller drivers	Network device drivers

FIGURE 2.7: Our Component Category for Linux Kernel

Linux Kernel Map [56] also provides us 411 keywords such as related variable/type names or directory/file names for the 37 components. These keywords will be used in the collection of component files.

2.5.2 Collect Component Files

We are going to identify which files each component includes. The Linux Kernel source code database we choose is maintained by Bootlin [57], and can be accessed through URL `https://elixir.bootlin.com/linux/$Version/$Type/$keyword`. Here the version is specified by variable `$Version`, variable `$Type` can be `"source"` to direct to a directory/file or `"ident"` to search for a variable/type and variable `$keyword` is the directory/file name or searched target.

Although different CVEs may affect different versions of Linux Kernel (the last version is version 5.1.5), we only choose the last version as our target because a later version usually contains more complete files.

For every component, we use the keywords mentioned in the previous sub-section that belong to it to crawl data from the above URL. There are three different cases:

- **Keyword is a file name**:

  If the keyword is the name of a `.c` file, we just record down the keyword.

- **Keyword is a variable/type name**:

  In this case, the URL to be requested is
  `https://elixir.bootlin.com/linux/latest/ident/$keyword`,
  and it provides a list of files that define or reference this keyword. File names are stored in `<strong>` tags and can be crawled by `soup.find_all('strong')`.

- **Keyword is a directory name**:

  The requested URL is
  `https://elixir.bootlin.com/linux/latest/source/$keyword`.
  It directs to a webpage that shows files and subdirectories in a directory whose name is the keyword. We extract all the file names by `soup.find_all('a', attrs='class' : re.compile("tree-icon icon-blob"))` since they are stored in `<a>` tags with string "tree-icon icon-blob" included in the class name. For the subdirectories, we use `soup.find_all('a', attrs='class' : re.compile("tree-icon icon-tree"))` to locate them and do a recursively traversal to obtain all contained files in every level of subdirectories.

Through crawling with the 411 keywords for the 37 component categories, we totally collect 30,724 Linux Kernel files in the latest version (5.1.5). The number of files in each component will be shown in the next section.

2.6 Results and Discussion

We collect 2162 Linux Kernel CVEs. The collection result distributed by published year is shown in Figure 2.8.

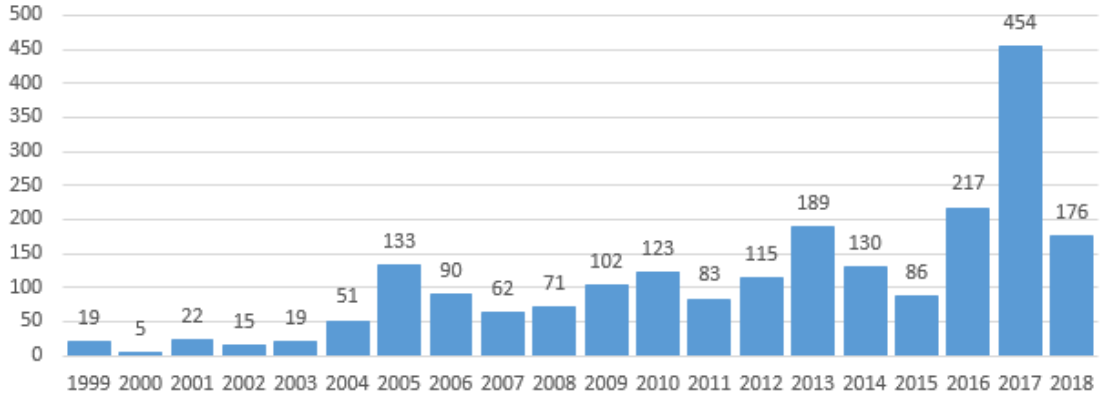


FIGURE 2.8: Number of Collected Linux Kernel CVEs with Years

We can see the yearly amount of publicly reported Linux Kernel CVEs are lower than 30 before 2004, and rapidly increase to 51 in 2004. Only after one year, in 2005, the amount leaps to 133. After then, the yearly reported amount generally shows an increasing trend and reaches the maximum value 454 in 2017. But one year after, the amount fell back to 176.

There are 8 different vulnerability types in our collected Linux Kernel CVEs: **Gain privileges**, **Denial Of Service**, **Bypass a restriction or similar**, **Overflow**, **Obtain Information**, **Execute Code**, **Memory corruption**, and **Directory traversal**. These types are categoried based on kinds of the consequence, while the criterion of classification we introduce in Section 1.2.1 is the root cause. Their detail amount and percentage are shown in Table 2.2. The tendency chart of CVEs with types and years is demonstrated in Figure 2.9.

It is needed to note that in these 2162 CVEs, some may not be categorised into any type, while some may belong to multiple types, which will be counted multiple times. So the total amount in Table 2.2 is 2619 instead of 2162, but the percentage is still calculated with 2162 as the dividend.

Rank	Type	Amount	Percentage
1	Denial Of Service	1186	54.8%
2	Obtain Information	350	16.2%
3	Overflow	346	16.0%
4	Gain privileges	260	12.0%
5	Execute Code	241	11.1%
6	Memory corruption	124	5.7%
7	Bypass a restriction or similar	112	5.2%
8	Directory traversal	3	0.1%

TABLE 2.2: CVE Amount and Percentage with Types

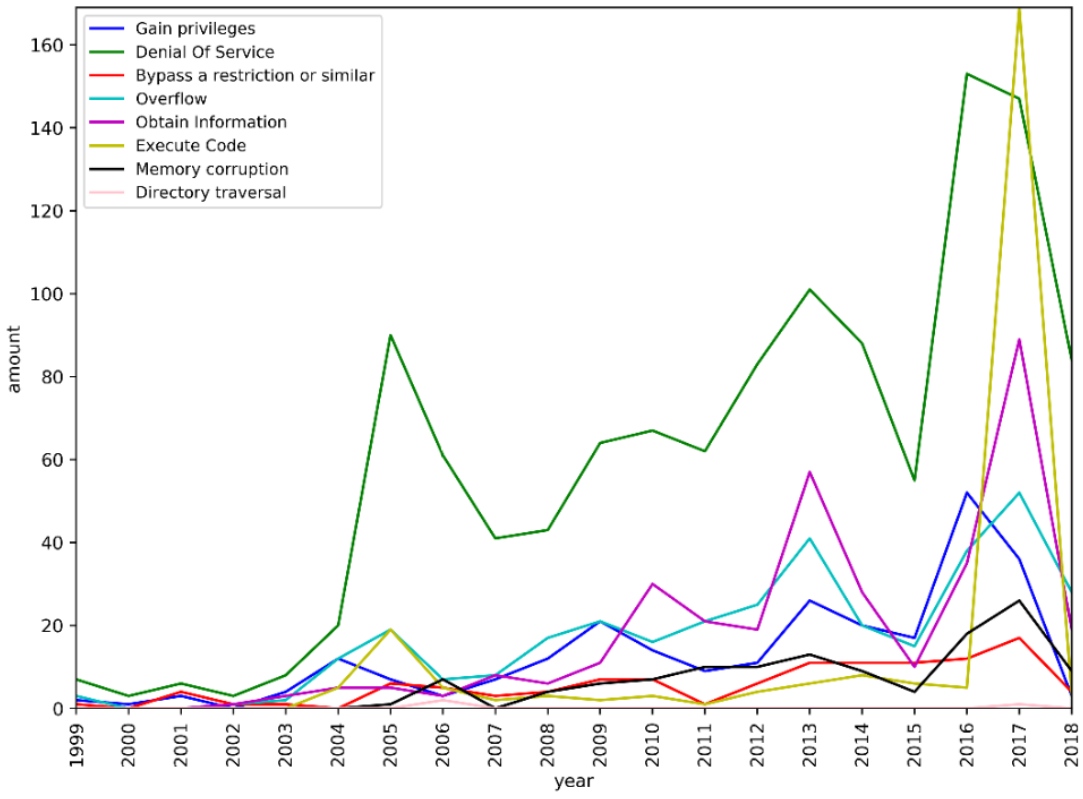
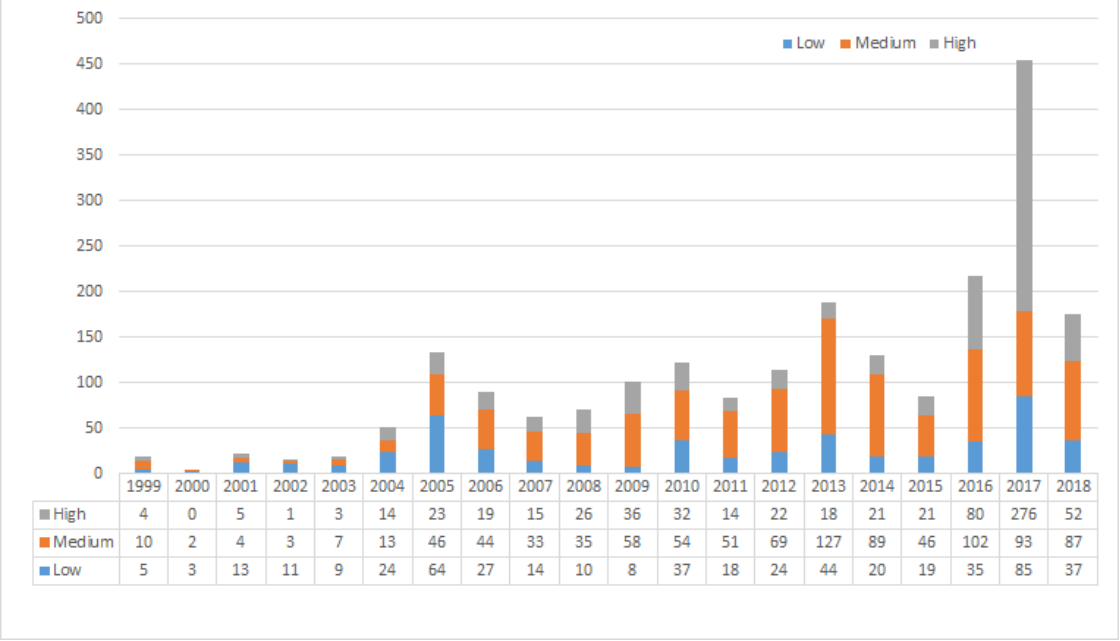


FIGURE 2.9: Number of Collected Linux Kernel CVEs with Types & Years

From the tendency chart, we can indicate that **Denial Of Service** is the most common vulnerability type that occurs in Linux Kernel. It stays at a high amount level (more than 40) since 2005 and always occupies the number one position except 2017. Amounts of CVEs in **Gain privileges**, **Overflow** and **Obtain Information** over the past 20 years also take high percentage (all above 12%). The rarest type is **Directory traversal**, which only occurs twice in 2006 and once in 2017. The trend of **Execute Code** is very interesting because of its explosive growth in 2017 due to a series of critical arbitrary code execution bugs found in Andriod products based on Liunx Kernel.

Another information we collect about Linux Kernel CVEs is the CVSS score, which is an industry metric to rate vulnerable levels. We count every year's CVE amount with different vulnerable levels(CVSS Score 0-4: Low, 4-7: Medium, 7-10: High) and demo the data in Figure 2.10.



	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018
High	4	0	5	1	3	14	23	19	15	26	36	32	14	22	18	21	21	80	276	52
Medium	10	2	4	3	7	13	46	44	33	35	58	54	51	69	127	89	46	102	93	87
Low	5	3	13	11	9	24	64	27	14	10	8	37	18	24	44	20	19	35	85	37

FIGURE 2.10: Number of Collected Linux Kernel CVEs with Scores & Years

It can be seen that CVEs in **Low** and **Medium** level constitute the major parts in every year except 2017. It is because of the same reason with the explosive growth of CVEs in **Execute Code** type: most of the **Execute Code** vulnerabilities reported in that year are able to cause serious damage to the system and are rated with a very high score (7-10). We should also pay attention to the growing trend

Data	Amount
Linux Kernel Components	37
Source Files in Components	30724
Linux Kernel CVEs	2162
Patch Commits	1310
CVEs with Patch Commits	1189
Files Modified in Patch Commits	2492
CVEs with Modified Files	1127

TABLE 2.3: Amount of Collected Data Related to CVE-Component Mapping

of high-risk vulnerabilities in the recent few years to prepare well for upcoming threats.

For CVE to components of Linux Kernel mapping work, we summarize the related data amount we collect in Table 2.3.

We collect 30,724 source files that are included in these 37 Linux Kernel components. The distribution of source files in components is shown in Figure 2.11. Since a source file can be categorized into multiple components, the summation of amounts in Figure 2.11 is much bigger than 30,724. We can see the top three components with maximum amount of files included is **synchronization** (12173), **Device Model** (9922) and **Logical memory** (8494).

There are only 1127 CVEs whose patch commits are accessible. We collect all files modified in patch commits for each of these CVEs, then map these 1127 CVEs to 37 components through the algorithm presented in Algorithm 2.

Figure 2.12 shows the mapping result. A CVE may also be mapped to multiple components according to the modified files in its patch commits. From the figure we find the top three components with maximum CVEs mapped to are **logical memory**(802), **synchronization**(726) and **threads**(620).

We also find out the top 5 most vulnerable Linux Kernel source files upon how many CVEs are relevant to it and show them in Table 2.4.

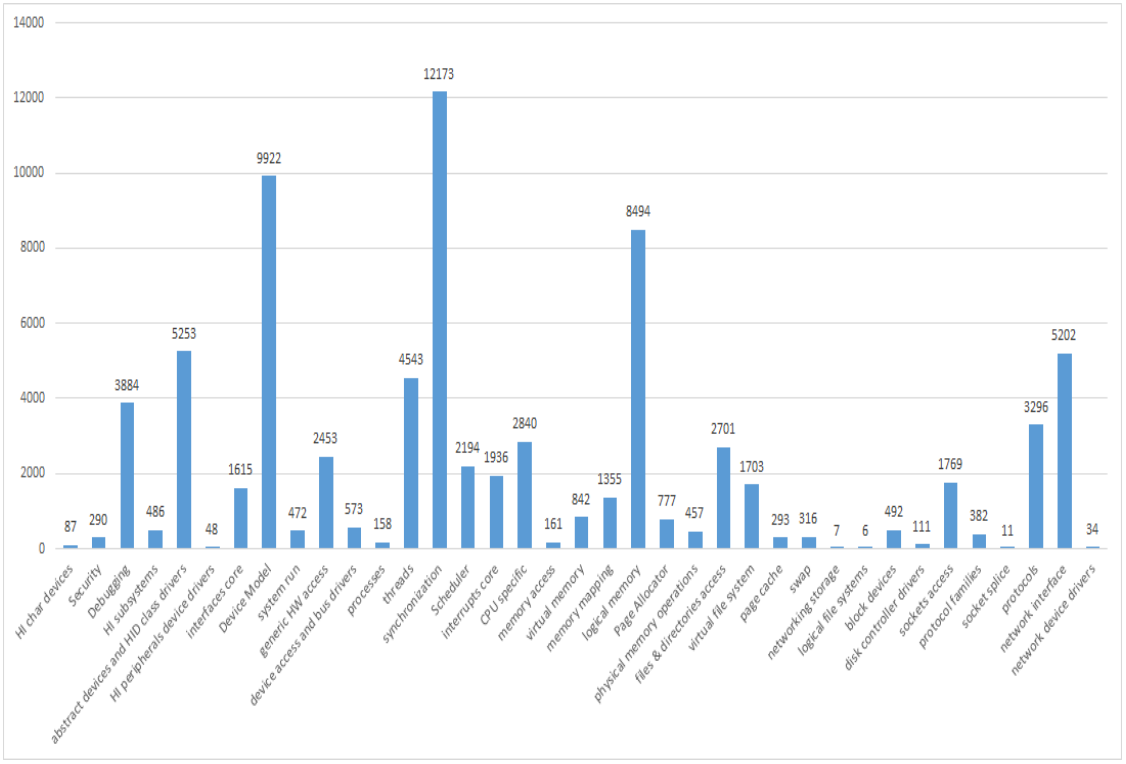


FIGURE 2.11: Amount of Source Files in Components

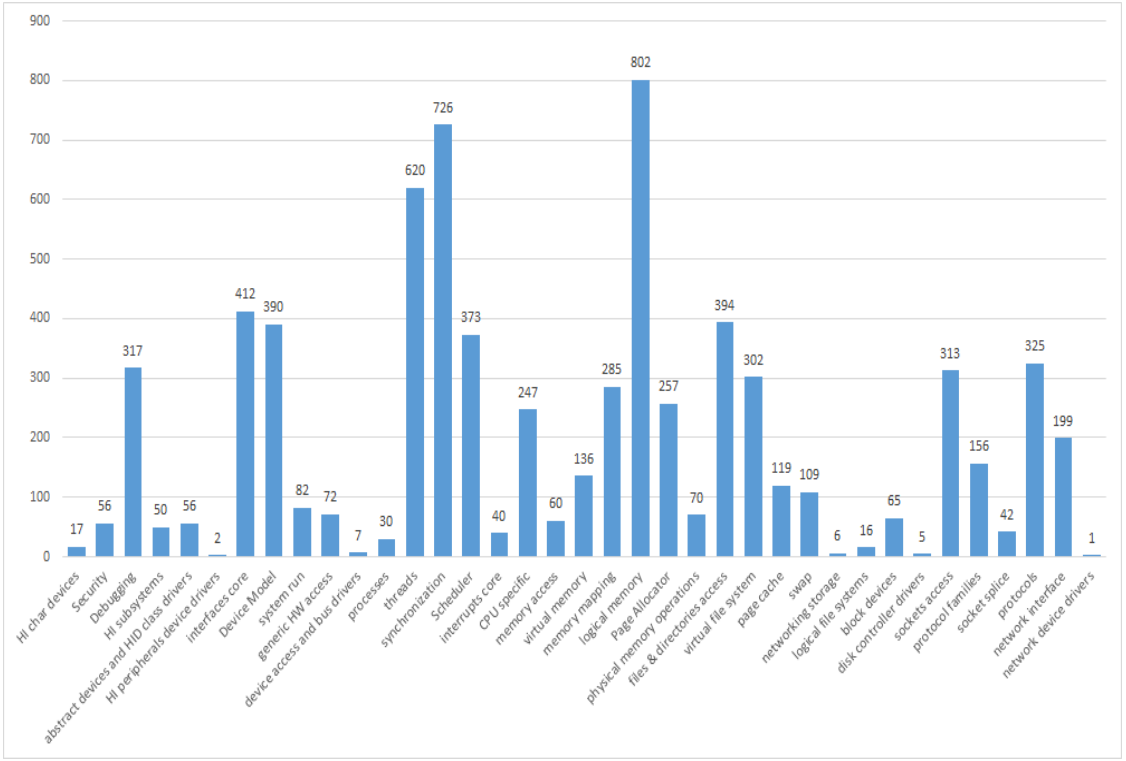


FIGURE 2.12: Amount of CVEs in Components

Algorithm 2 CVE-Component Mapping Algorithm

Input: Dictonary of CVEs with Modified Files, D_V; Dictonary of Components with Included Files, D_C;

Output: Dictonary of Components with mapped CVEs, D_{VC}

```
for cve in D_V.keys() do
    for comp in D_C.keys() do
        if Any match pairs between D_V[cve] and D_C[comp] then
            D_VC[comp].add(cve)
        end if
    end for
end for
```

File	Relevant CVE Amount	Relevant Component Amount
arch/x86/kvm/x86.c	17	8
net/socket.c	16	14
fs/ext4/super.c	14	10
kernel/bpf/verifier.c	14	9
arch/x86/kvm/vmx.c	14	8

TABLE 2.4: Top 5 Linux Kernel Source Files upon Relevant CVE Amount

We calculate the Files/CVEs ratio for each component and show the result in Figure 2.13 (set the value to zero for components with no CVEs). Smaller value (except for zero) indicates more vulnerable. The component with the smallest ratio is **socket splice**, which is found with 42 CVEs and with only 11 files included. The second vulnerable component is **logical file system**, which has 16 CVEs found and 6 CVEs included. If we only consider components that have more than 300 CVEs, the most vulnerable one is **interfaces core** with 412 CVEs, 1615 files, and the ratio is 3.92. Other low ratio components includes **socket access** (5.65), **Scheduler** (5.88) and **file & directories access** (6.86). Component **Synchronization** contains the maximum amount of files but its Files/CVEs ratio is 16.77. And component **logical memory**, which gets the maximum amount of CVEs found in, has a ratio of 16.77.

Through summarizing the data shown in Figure 2.11, Figure 2.12 and Figure 2.13,

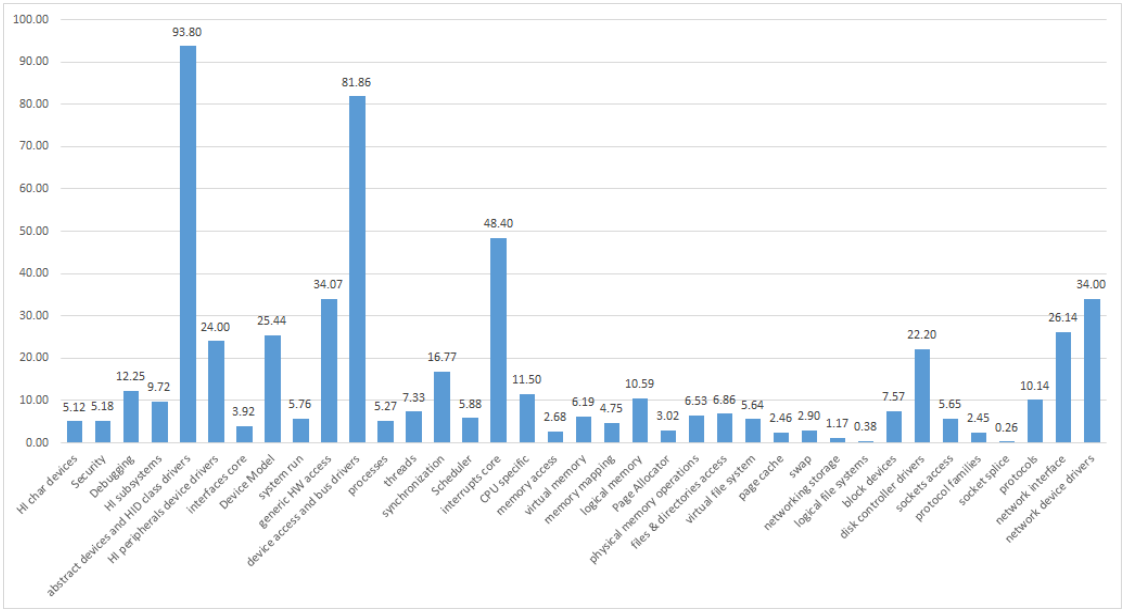


FIGURE 2.13: Component Files/CVEs Ratio

we plot the chart of the vulnerable level for each component in Figure 2.14. Here **Few CVEs**, **Many CVEs** and **Massive CVEs** denote the CVE amount is less than 100, more than 100 but less than 300, and more than 300. Demarcation point of **High and Low Ration** is 10.00. From that we can easily figure out that all the Linux Kernel components inside layers of **user space interface**, **Virtual**, **Bridges** and **Logical** are at a high vulnerable level, while the **Device control** and **Hardware interfaces** layer are much more safer.

Func / Layers	Human interface	system	processing	memory	storage	Networking
user space interface	HI char device	Interfaces core	Processes	Memory access	Files & directories access	Sockets access
Virtual	Security	Device Model	Threads	Virtual memory	Virtual file system	Protocol families
Bridges	Debugging		Synchronization	Memory mapping	Page cache / Swap	Networking storage / Socket splice
Logical	HI subsystems	System run	scheduler	Logical memory	Logical file systems	protocols
Device control	Abstract devices and HID class drivers	Generic HW access	Interrupts core	Page allocator	Block devices	Network interface
Hardware interfaces	HI peripherals device drivers	Device access and bus drivers	CPU specific	Physical memory operations	Disk controller drivers	Network device drivers

Few CVEs High Ratio | Few CVEs Low Ratio | Many CVEs High Ratio | Many CVEs Low Ratio | Massive CVEs

FIGURE 2.14: Vulnerable Level of Linux Kernel Components

In conclusion, we searched and collected Linux Kernel CVE data such as ID, type, score, patch commits and relevant changed files. We also collected sets of Linux Kernel files for 37 components categorised by Constantine's Linux Kernel map [56]. Then we built a mapping between Linux Kernel CVEs and components through matching source files in both of them. Our mapping work identifies vulnerable levels of these 37 components and can be used to benefit the detection of Linux Kernel vulnerabilities in the target identification aspect. Our approach is also extensible with new CVE data and a more advanced classification strategy of the target component.

Chapter 3

Code Coverage Improvement for Coverage Guided Fuzzing

3.1 Background

3.1.1 Code Coverage

Code coverage is a crucial and widely used metric in software testing. It presents which part of the source code is executed during the execution of a particular test case, and measures how many percentages of coverage achieved by this test case. Test cases with higher code coverage are usually regarded as having a higher chance to discover undetected bugs compared to those with lower coverage.

There are different criteria for different aspects of code coverage, mainly including statement coverage, function coverage, basic block coverage, and edge coverage.

We use a simple C function in Listing 3.1 to explain the basic concepts of them.

```
1 int foo(int k){
2 if (k)
3 k = 0;
4 return k;}
```

Listing 3.1: A Sample C Function

- **Statement Coverage**:

  Also known as line coverage since usually one statement takes one line. It identifies statements executed in a run of test case. In the above C function, **foo(1)** will execute all the staments while **foo(0)** cannot execute **k=0;**.

- **Function Coverage**:

  Coverage is measured on function level. If a function is called during execution, mark it as covered for the executed test case. For example, if function **foo** in Listing 3.1 is called, it will be listed in the covered functions in function coverage.

- **Basic Block Coverage**:

  The code of a program can be decomposed into basic blocks that contain only one entry point and one exit point, with no branches in the middle. In other words, every node in a control flow graph denotes a basic block. Basic Block Coverage measures which basic block is hit in an execution.

  A function may contain multiple basic blocks, for example in Listing 3.1, the function **foo** contains three basic blocks: statement **k = 0;** forms **basic block B**, statement **return k;** forms **basic block C** and rest forms **basic block A**. If we call **foo(1)**, all basic blocks **A, B** and **C** are hit. If call **foo(0)**, clearly that basic block **B** will not be hit.

- **Edge Coverage**:

  A jump from one basic block to another creates an edge. Still consider the function in Listing 3.1, **foo(1)** results in a path with two edges: A→B→C, while **foo(0)** will only trigger one edge: A→C.

  Edge coverage tells us which edge is triggered during an execution. It is more accurate than function coverage and already subsumes covered basic blocks. Furthermore, it can provide more detailed control flow information than basic block coverage. Compared to statement coverage, edge coverage is able to be obtained in grey or block box testing. It is also able to discover issues arising in control flow constructs through identifying state transitions.

3.1.2 Coverage Guided Fuzzing

Coverage guided fuzzing is a fuzzing approach guided by code coverage information. Here the code coverage produced by an existing test case is used as feedback to evaluate its value. A test case is regarded as valuable or interesting if it triggers new coverage. Only valuable test cases will be kept for mutation and further fuzzing.

New coverage is determined upon which criterion introduced in Section 3.1.1 is used. The most broadly used criterion is *edge coverage*, in which a test case obtains new coverage if there is at least one edge never observed before being triggered by it. For example, with the sample control flow graph shown in Figure 3.1, in case that the path A→B→C→D has already been observed, a test case that triggers path A→B→A→B→C→D obtains a new edge coverage because it triggers an unobserved edge **BA**. Another test case that triggers path A→B→D is also considered as interesting since the new edge **BD** is discovered. After that, a test case that triggers A→B→A→B→D won't be kept due to all edges in this path have been triggered before.

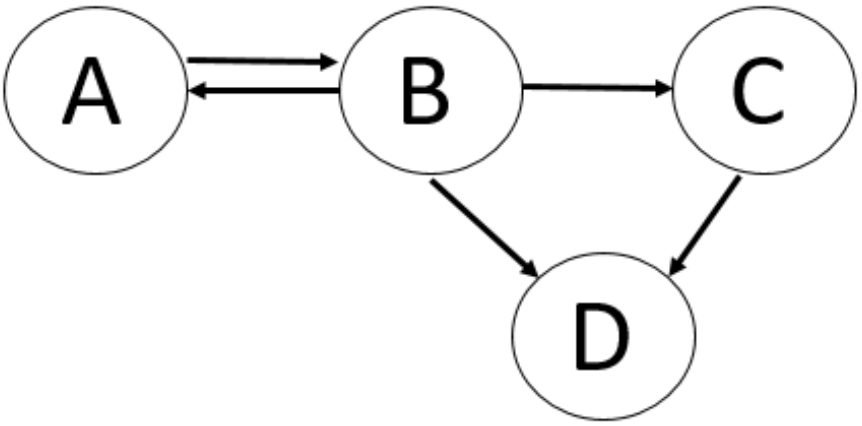


FIGURE 3.1: A Sample Control Flow Graph

In recent years, coverage guided fuzzing demonstrates its high efficiency in the practice of vulnerability detection. Many fuzzing frameworks [28, 32, 58] inspired by this guided approach have been developed and several works [15, 33, 59] were proposed to improve the performance of them. Here we discuss some of these works:

American Fuzzy Lop (AFL) [28] is one of the most popular coverage guided fuzzing frameworks. It regards the transition between basic blocks as coverage information and obtains them through applying compile-time instrumentation. New test cases are generated via mutation on existing test cases in an input queue. A test case that triggers a new transition is marked as interesting and will be saved into the queue. Therefore, AFL is able to gradually increase the total coverage of fuzzing target, execute more code regions thus detect more vulnerabilities.

Steelix [59] provides a boosting technique to AFL in the aspect of exploring paths protected by comparisons of magic bytes. Based on AFL, it applies extra static analysis and binary instrumentation to fetch comparison progress information, then utilizes such information to identify the location of magic bytes as well as how to reach the correct magic bytes. With that, it helps the fuzzer to penetrate more paths and achieve higher coverage.

FairFuzz [15] is a tool also based on AFL. It counts the number of hits for each branch discovered during the fuzzing, then marks branches that are hit few times below a pre-defined rarity-cutoff value as rare branches. Instead of mutating all test cases in the input queue, FairFuzz will only mutate these hitting at least one rare branch. And the mutation strategy is modified to ensure new test cases still be able to hit a given rare branch. In one word, it optimizes the distribution of fuzzing energy via spending more energy on code regions that are rarely touched.

3.2 Motivation

Recently coverage guided fuzzing demonstrates its outstanding performance in software vulnerability detection. But we notice that the most popular coverage guided fuzzing frameworks, such as AFL, are all based on the mutation approach. One reason is that in case an interesting sample is given as seed, test cases generated by mutation are more related to the seed and can inherit the interesting features more easily. Another reason is that the basic mutation-based approach requires less knowledge about the structure of test cases since mutative strategies such as bit

or byte flipping can be implemented randomly, thus make the fuzzer being generic to suit different targets.

However, a test case that cannot bypass syntactic and semantic check processed by the target is an invalid input and regarded as useless for triggering real bugs. These mutation-based fuzzing frameworks usually generate invalid test cases and waste many time on that, causing these fuzzers to be inefficient when the target requires highly-structured inputs.

By contrast, generative fuzzing frameworks perform better when facing challenges from highly-structured inputs. Because generative fuzzing is able to generate test cases according to the given syntax features and ensure these test cases are syntactic valid. And some widely used generative fuzzers utilize manually-summarized and specified semantic rules as well as code emulation to improve the probability of semantic valid for a certain fuzzing target, such as Domato [25] for DOM structure and jsfunfuzz [60] for JavaScript engines.

Driven by above, in this work, we explore the possibility of applying coverage guided fuzzing strategy on generative fuzzing framework to achieve both benefits on fuzzing applications with high-structured inputs. We choose the HTML DOM structure in Chromium browser [61] as the fuzzing target, and Domato [25] as our test case generator to generate .html samples. We obtain edge coverage during execution through a code coverage instrumentation provided by SanitizerCoverage [62]. We use the rarely-hit edge as a metric to judge whether a test case is interesting and implement a coverage-guided approach. We hope to observe a significant growth of edge coverage for the test cases generated upon interesting samples.

3.3 Approach

3.3.1 Overview

We would like to present our approach shown in Figure 3.2 through comparing with the widely-used classical mutation-based coverage guided fuzzing approach shown in Figure 3.3.

The classical approach in Figure 3.3 maintains an input queue. The fuzzer iterates the queue and implements pre-defined mutation strategies on current proceeding input to generate more samples. It executes these mutated samples on fuzzing target and crashes are detected by an attached monitor. After execution, the code coverage is computed through instrumentation on target during its compiling time. Then the fuzzer launches a judgement on the coverage result. If the result triggers any new behavior, such as hitting a new basic block or branch, depending on which metric is used, the sample is regarded as an "interesting sample", and after some refinement operations it will be appended to the input queue. As the queue growing, more interesting samples are used as bases for mutation and the code coverage increases gradually, resulting in a higher chance to find out vulnerabilities.

Our approach in Figure 3.2 is different from the classical one in some aspects. Firstly, since it is generation-based, the queue we maintain should contain templates for generating samples instead of samples themselves, then in each iteration, Domato generator takes the current template to generate a massive amount of .html samples. Secondly, instead of using only newly undiscovered edges as metric of judgement, we regard samples that hit an edge which is hit very rarely as interesting, because an edge is rarely hit means there is a code region which is rarely explored, leading to a higher probability to trigger new further edge starts from this region. To identify rare edges, we build a database that records all the discovered edges as well as their number of hit. Then we find out samples that hit at least one rare edge, refine, minimize and format them into templates while keeping the condition that rare edge gets hit. Finally we append these templates into the queue for further generation.

3.3.2 Fuzzing Target

Our fuzzing target in this work is the DOM structure in Google Chromium browser on Linux platform.

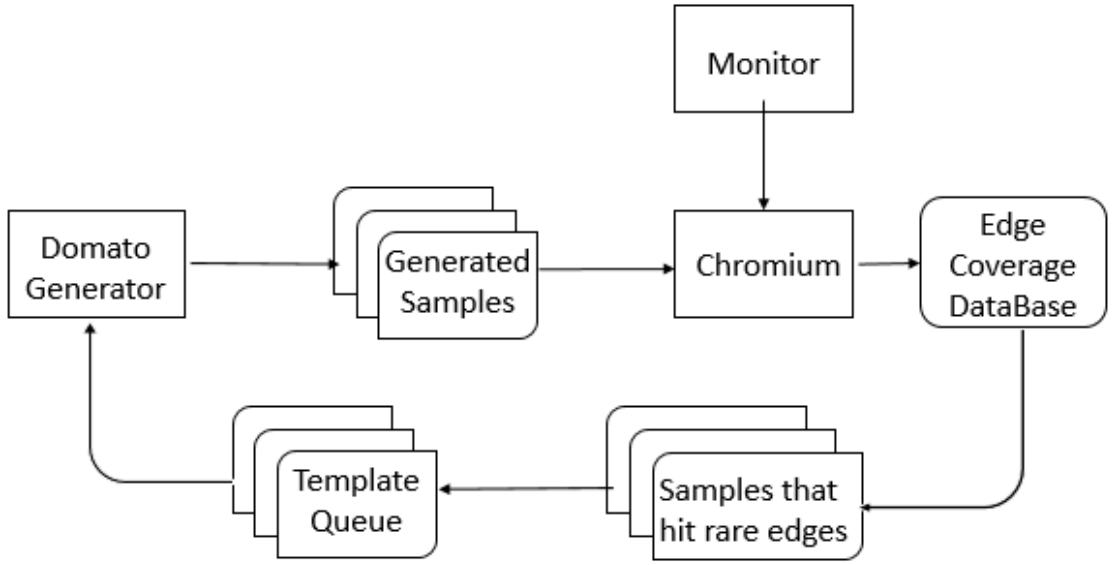


FIGURE 3.2: The Overview of Our Approach

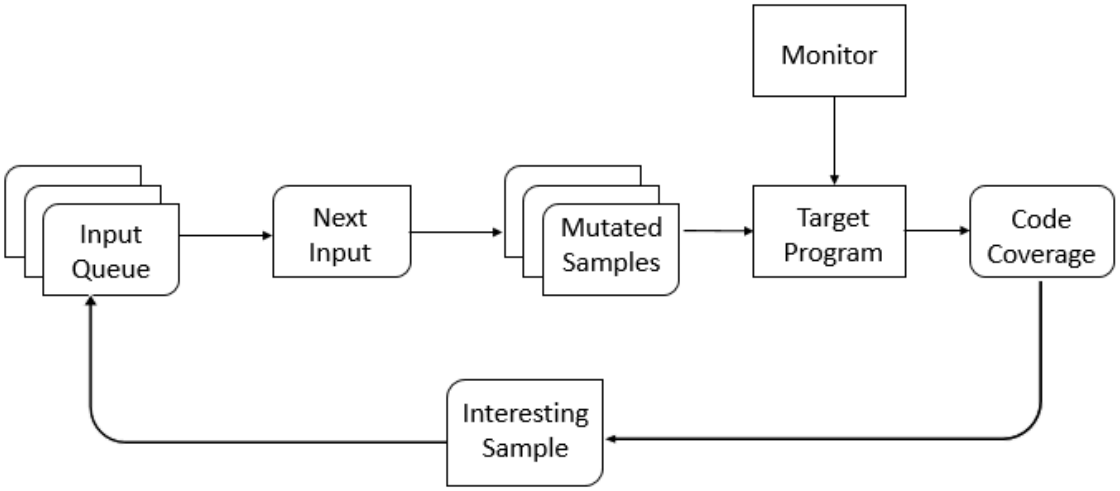


FIGURE 3.3: Classical Mutation-based Coverage Guided Fuzzing Approach

DOM presents for "Document Object Model", which is an application programming interface that treats HTML or XML documents as a logical structure, commonly like a logical tree. It defines how a document can be accessed and manipulated, allows these documents to be used in object-oriented programs. In past decades, DOM was specified standardly and supported by the most widely used browsers. Therefore, DOM becomes one of the most common attack vectors for browser fuzzing.

Google Chromium is an open-source web browser and many vendors have utilized

its source code as the basis to develop their own browsers such as Maxthon and 360 secure browser. It suits Linux platform well and can be instrumented during compiling time for crash monitoring and coverage computing purpose.

3.3.3 Test Case Generator

Our test case generator is based on Domato [25], a DOM fuzzer that provides a grammar-aware generative approach to generate .html files, which can be parsed into a DOM tree by browsers. Its brief workflow is shown in Figure 3.4.

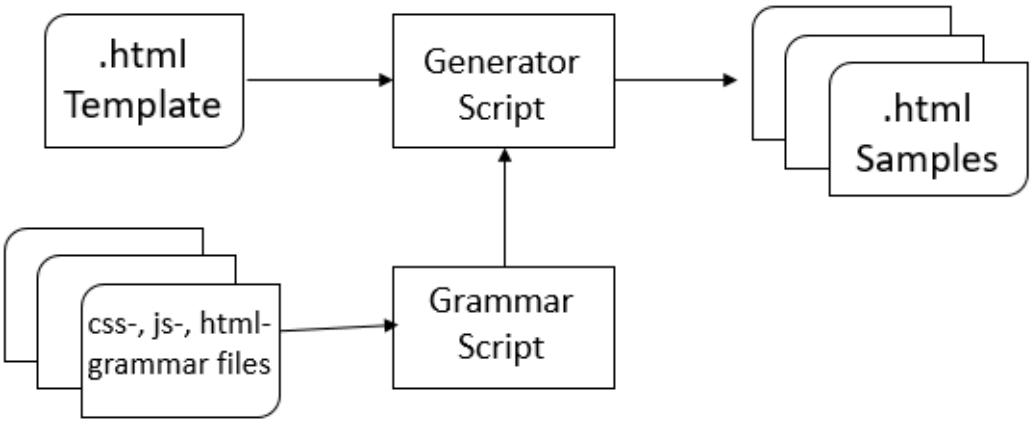


FIGURE 3.4: Workflow of Samples Generation

The generator consists of four components: a .html template, a series of grammar files, a grammar script and a main generator script.

- **Template**:

  The template for sample generation is a .html file where the content can be programmatically accessed by HTML DOM methods. Listing 3.2 shows its structure.

```
<html>
  <head>
    <style>
      <cssfuzzer>
    </style>
    <script>
      function jsfuzzer() {<jsfuzzer>}
      function eventhandler*() {<jsfuzzer>}
    </script>
  </head>
  <body onload=jsfuzzer()>
     <htmlfuzzer>
  </body>
</html>

```

Listing 3.2: Structure of Template

There are three components that can affect the performance of the template and need to be generated: Hyper Text Markup Language(HTML), Cascading Style Sheets(CSS) and JavaScript. The template uses `<htmlfuzzer>`, `<cssfuzzer>` and `<jsfuzzer>` to mark them and specifies where to generate them.

Do note that inside the `<script>` tag, the template first defines a JavaScript function **jsfuzzer()** which will be triggered once the page has been loaded, then it defines multiple functions named **eventhandler*** (by default * ranges from 1 to 5, means in total 5 handler functions) for responding to various events. JavaScript code will be generated for each of these functions. And every JavaScript function is designed to be called at most twice, aims to prevent the template from infinite calling loops.

- **Grammar Files**:

  Generation of HTML, CSS and JavaScript code follows their own grammar. Domato generator obtains these grammar through both browser code extraction and manual edition. Grammar for these languages used by this generator is expressed in a set of the following format:

```
<symbol> = a mix of constants and <other_symbol>s
```

It means the symbol on the left side can be expanded by a rule defined on the right side, and symbols on the right side are also expanded recursively in their own corresponding rules. For example, following the CSS grammar given in Listing 3.3, when we generate the symbol "cssproperty", we will get "stop-color: var(–cssvara)".

```
<cssproperty> = <cssproperty_name>: var(<cssvar>)
<cssproperty_name> = stop-color
<cssvar> = --cssvara
```

Listing 3.3: CSS Grammar Example

For a symbol that has more than one expanding rules, we can specify the probability of each rule in the grammar file. Otherwise, the generator will choose a rule for that symbol randomly.

All grammar rules for generating a programming language code are enclosed in "!begin lines" and "!end lines". Each line of the grammar can be parsed and selected for generating specified lines of code.

- **Grammar Script**:

  Domato generator uses a Python script to parse grammar files which are in the format mentioned above and generate corresponding code.

  In this script, a class named **Grammar** is defined. Method **parse_from_file** of **Grammar** takes a grammar file as input and parse its content line by line to load the grammar rules. Then another method **_generate_code** is responsible for generating a requested number of lines of code based on the parsed rules.

  The grammar script implements a generic grammar-aware generation since it is applicable to any grammar files as long as expressed in the required format.

- **Generator Script**:

  Another Python script is used as the main engine for DOM sample generation. A simplified version of its generating algorithm is shown in Algorithm 3.

Algorithm 3 Algorithm to Generate Samples

Input: .html template: *template*; grammar files: *html.txt*, *css.txt*, *js.txt*;

Output: A set of generated samples: *outfiles*

```
1: import Grammar
2: htmlgrammar = Grammar()
3: htmlgrammar.parse_from_file('html.txt')
4: cssgrammar = Grammar()
5: cssgrammar.parse_from_file('css.txt')
6: jsgrammar = Grammar()
7: jsgrammar.parse_from_file('js.txt')
8: for outfile in outfiles do
9:     result = template
10:    result.replace('<cssfuzzer>', cssgrammar.generate_symbol())
11:    result.replace('<htmlfuzzer>', htmlgrammar.generate_symbol())
12:    result.replace('<jsfuzzer>', jsgrammar._generate_code())
13:    write(result, outfile)
14: end for
```

To generate .html samples, the generator script imports class **Grammar** from grammar script, and creates three instances of it: **htmlgrammar** for parsing and loading HTML grammar, **cssgrammar** for CSS and **jsgrammar** for JavaScript. After that, it generates each new sample iteratively by taking a specified .html whose format is mentioned in Listing 3.2 as template. The generator script will locate positions for HTML, CSS and JavaScript generation by searching pre-defined tags: <htmlfuzzer>, <cssfuzzer> and <jsfuzzer>. Then it replaces these tags with the corresponding language statements which are generated following their own grammar.

One thing to be noted is HTML and CSS must be generated before JavaScript, because the element information such as id and class name need to be saved for generating semantically valid JavaScript code.

3.3.4 Monitor

A monitor is in charge of monitoring the executive status of the target program during its fuzzing process. Once an unhandled crash arises, execution halts and monitor records the crash dump and saves it together with the sample which triggers this crash into a place specified by the fuzzer, called crash archive. Sometimes a powerful monitor also plays the role of crash analysis, it will categorise crashes according to the recorded memory states in their dump file, check further a crash as exploitable or not. Depending on the fuzzing target and fuzzing platform, a monitor can be attached to the fuzzing target process or directly integrated with it through instrumentations during compiling.

In our work, we use **AddressSanitizer** (ASAN) [63], an open source detector for addressability issues. It utilizes the shadow memory technique [64] to record the safety status of each byte in application memory, and raises a report once any unsafe bytes are accessed. AddressSanitizer has demonstrated its outstanding performance in bug detection on various browsers and is recommended by Chromium developers for testing purposes. So we build Chromium with ASAN by setting **is_asan** flag to be true in **args.gn** file. In this case, AddressSanitizer is used to instrument Chromium binaries and its monitored results can be written to a log file after the execution of a test case.

3.3.5 Coverage

We need to obtain coverage information of test cases after their execution. **SanitizerCoverage** [62] is used for this purpose.

SanitizerCoverage is a built-in feature of **clang** [65], the main compiler of Chromium building. It provides three flags for three different levels of instrumentation: edge(default) for edge-level, bb for basic-block-level and func for function-level. To instrument the target program with this feature and get edge-level coverage information, following flags are set in **args.gn** file:

use_sanitizer_coverage = true
sanitizer_coverage_flags = "trace-pc-guard"

When we execute "chrome", the main executable binary of Chromium, with a .html file as input and with flag "coverage" set in ASAN_OPTIONS, a file named "chrome.*.sancov" (* denotes the pid number) is created for dumping the edge coverage once the execution is completed. Since "chrome" is dynamically linked with many dynamic shared objects (DSO) during the run time, every DSO will have its own .sancov file created. We find that there will be around 157 to 160 .sancov files for one test case. Except "chrome.*.sancov" for the main program "chrome", all the other .sancov files are for the dynamic libraries postfixed by .so.

A .sancov file contains the data of all hit edges in its corresponding binary or DSO. These edge coverage data can be extracted by a script called *sancov.py*. Its usage is as the following:

```
sancov.py print <input .sancov file>
```

Since a single test case will get multiple coverage files created, we write another script to iterate through all coverage files and merge the edge coverage data into a dictionary for every test case. The dictionary is formatted with the following keys:

- **"id"**: A string denotes the ID of the current test case.
- **"file_len"**: An integer counts for how many binaries/DSOs includes.
- **binary/DSO name**: A list of strings that denote the covered edges in the current binary/DSO.

Every dictionary is stored in a .json file named by its test case id. After the execution of all the test cases, these .json files are merged together to create a coverage database. The database contains not only all ids and covered edges, but also a counter for every edge to count how many test cases trigger it during the execution. For example, if edge B in binary A is triggered by 100 test cases, the recording in our coverage database should be **'A': { 'B': 100 }**.

3.3.6 Scoring

It is important to properly score the test cases by a metric in guided fuzzing. A test case which performs an unusual behavior will gain a high score. Mutation-based coverage guided fuzzing frameworks universally define the "unusual behavior" as triggering a new coverage that has never been discovered before, here the "new coverage" is used as the metric of scoring.

In contrast to choosing **"new edge"**, we choose **"rarely-hit edge"** as the metric. The reason is firstly, without high-quality directed techniques integrated, the generative test cases are often stuck in a big difficulty of discovering new edges with a massive coverage database already built. Secondly, if an edge is rarely hit by existing samples, means the code region directed by it is rarely exercised, then generation based on samples which hit this edge provides a higher probability to explore the rarely executed code region in more detail, resulting in a higher chance to trigger new coverage and detect deep-hidden bugs.

To define what is a rarely-hit edge, let T be the set of existing test cases, E be the set of covered edges found by elements in T. So we have the number of hits for an edge e in E calculated by:

$$hitCount[e] = |\{t \in T : t \; hits \; e\}|$$

As described in Section 3.3.5, the counter of hit times for every edge is all stored in our coverage database and can be obtained through querying with the binary id and edge id (the data printed by *sancov.py*).

Now let E_r be the set of rarely-hit edges, then E_r can be obtained by:

$$E_r = \{ \; e \in E : hitCount[e] \leq threshold \; \}$$

Here the value of *threshold* should be determined according to a comprehensive consideration on hitting status of the edges, and it needs to be updated as the database changes.

Finally, we get T', the set of test cases that hit at least one element in E_r, computed through:

$$T' = \{t \in T : \exists\ e \in E_r,\ e\ is\ hit\ by\ t\}$$

All the elements in set T' are marked as "interesting" upon our scoring metric and chosen as candidates for the process of refinement described in the next subsection.

3.3.7 Refinement

By scoring, we get a set of interesting test cases. But before adding them to the queue of generative templates, it is necessary to perform a refinement. The purposes of refinement include two aspects: in one hand, ensure the interesting behavior of a test case is deterministic reproducible, and in the other hand, minimize the size of test cases by removing redundant statements and keeping the interesting behavior unchanged.

- **Check reproducible**

  If the interesting behavior disappeared in the second run of a test case, the behavior may be thought as a rare coincidence caused by some uncertain and uncontrollable factors, resulting in a very small chance to reproduce itself. Only test cases that are able to perform interesting behaviors in a deterministic way are valuable to save as templates. That is why we need to perform a reproducible check.

  Since **"interesting behavior"** is defined as **"hitting at least one rarely-hit edge"** in our work, a test case is thought as performing a reproducible interesting behavior if it is able to hit at least one original rarely-hit edge in multiple rounds of execution. The reproducible checking algorithm is shown in Algorithm 4.

Algorithm 4 Algorithm for checking reproducible

Input: Test case to check: t; A set of rarely-hit edges hit by t: E; Rounds of check: N; Target program: P

Output: A boolean value indicates reproducible or not: R

```
1: R = True
2: for 0 ≤ i < N do
3:     E' = Get_Edge_Coverage(t,P)
4:     if E' ∩ E = ∅ then
5:         R = False
6:         return R
7:     end if
8: end for
9: return R
```

In this checking algorithm, once the covered edge set of any running round has no union with the rarely-hit edge set gotten in Section 3.3.6, it will return *False*. It returns *True* if and only if every round's coverage hits at least one rarely-hit edge. That ensures the test case performs interesting behaviors in a deterministic way.

- **Minimization**

  In order to obtain as much code coverage as possible, the original test cases are usually generated in a quite big size. If we straightly use them as templates, the further generated test cases will be in a double size but can not achieve an obvious growth in coverage, causing a drastic decrease in performance of further fuzzing. To prevent that, minimization on interesting test cases is important and necessary for coverage guided fuzzing, especially for the generation-based approach.

  Minimization on a test case aims at reducing its size while keeping its interesting behaviors unchanged. A straightforward method of that is to remove redundant statements as much as possible. If a statement has no effect on a specified behavior, it is regarded as unnecessary and redundant for this behavior. For example, in Listing 3.4, to trigger the interesting behavior in line 4, the assignment in line 1 is necessary for satisfying the if-statement in

line 3, but the assignment in line 2 is redundant since interesting_behavior() can still be touched in case of its removal.

```
1  a = 1;
2  b = a*2;
3  if (a*2)
4     interesting_behavior();
```

Listing 3.4: Redundant Code Example

In our work, we propose an indefinite iterative approach shown in Algorithm 5 to minimize a test case, its simplified workflow is also shown in Figure 3.5

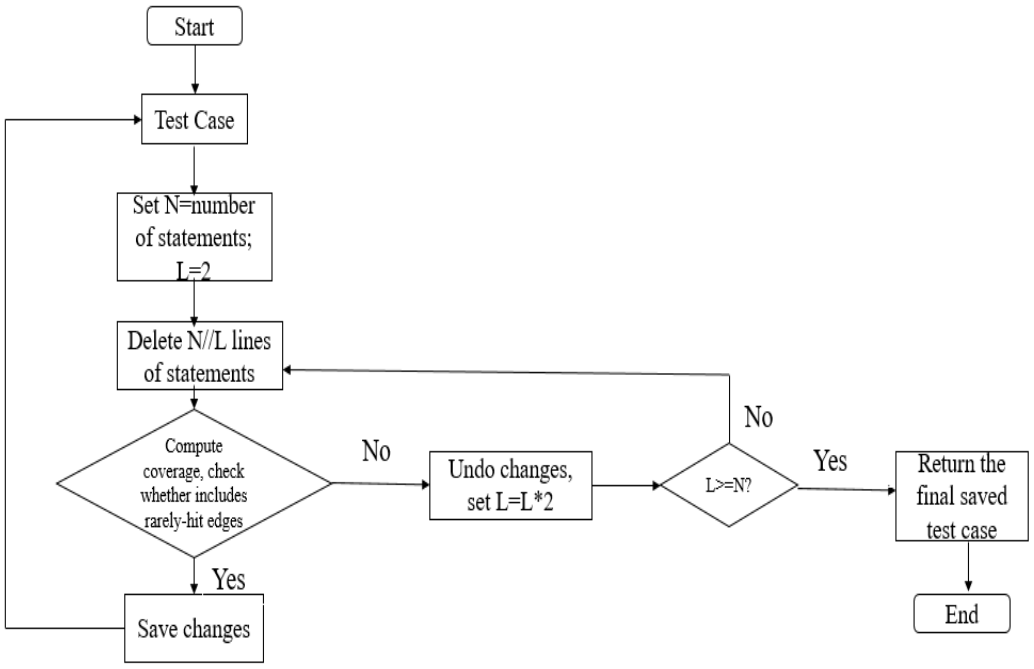


Figure 3.5: Workflow of Minimization

In each iteration, we fist figure out the number of removable statements in the current sample, which include all the HTML elements, CSS rules and JavaScript codes generated by our generator, represented by N, and set a level counter L initialized as 2, as well as a counter C for try times of statement removal under current level.

Algorithm 5 Algorithm for Minimization

Input: Test case: t; Rarely-hit edges set hit by t: E; Target program: P

Output: The final minimized test case

```
1: while True do
2:     N = Count_Statements(t)
3:     L = 2
4:     C = 0
5:     while True do
6:         t' = Remove_Statements(t,N//L)
7:         E' = Get_Edge_Coverage(t',P)
8:         if E' ∩ E ≠ ∅ then
9:             t = t'
10:            break
11:        else
12:            C += 1
13:            if Counter ≥ L/2 then
14:                L = L * 2
15:                C = 0
16:            end if
17:            if L ≥ N then
18:                return t
19:            end if
20:        end if
21:    end while
22: end while
```

Then in the secondary while-loop, remove $N//L$ lines of statements to get a temporary sample version and check its edge coverage. If its coverage includes any rarely-hit edges hit by the original sample, save the temporary one and break out of the secondary while-loop. Otherwise, check whether C reaches the limitation of try times for the current level L by comparing C with $L/2$. Observably, bigger L leads to a higher limitation of try times as well as a smaller number of statements to remove. Once C reaches the limitation, double the L value and zero C, use them in the next iteration

of the secondary while-loop unless $L \geq N$, which indicates we tried enough times and can say there are no redundant statements left.

Here function $Remove\_Statements(t, N//L)$ is used to randomly remove $n = N//L$ lines of statements in current sample t. As shown in Algorithm 6. A list of removable statements is extracted through matching with regular expressions in Listing 3.5. Then iteratively select a random index and remove the statement at that index for n times.

Algorithm 6 Function *Remove_Statements*()

```
1: function REMOVE_STATEMENTS(t, n)
2:     Statement_List = Get_Statements(t)
3:     for 0 ≤ i < n do
4:         index = random(0, len(Statement_List) − 1)
5:         t = t.replace(Statement_List[index],'')
6:     end for
7:     return t
8: end function
```

```
JavaScript: '(?<=//beginjs\n)(?:.|\n)*?(?=//endjs\n)'
CSS: '(?<=/\*begincss\*/\n)(?:.|\n)*?(?=/\*endcss\*/\n)'
HTML: '(?<=<!--beginhtml-->\n)(?:.|\n)*?(?=<!--endhtml-->\n)'
```

LISTING 3.5: Regular Expressions to match JavaScript, CSS and HTML

After refinement, we can get a set of minimized test cases that are able to deterministically trigger at least one rarely-hit edge. These test cases still follow the format of our original template since we only remove some newly generated HTML, CSS and HTML statements. We re-add `<htmlfuzzer>`, `<cssfuzzer>` and `<jsfuzzer>` tags to them so that they can be used as new templates and feed to Domato generator for further generation. Then we will generate new test cases based on our interesting templates and evaluate their coverage improvement.

3.4 Implementation and Evaluation

As explained in Section 3.3, we choose Chromium [61] version 73.0.36633.0, built with arguments shown in Figure 3.6 as our fuzzing target. Choose Domato [25] up to commits dd898d3 on Jan 31, 2019 as our test case generator. The monitor is integrated with fuzzing target through a compiler instrumentation provided by AddressSanitizer [63] implemented in Clang 8.0.0 [66]. Code coverage is also obtained by a clang's built-in instrumentation called SanitizerCoverage [62].

```
# Build arguments go here.
# See "gn args <out_dir> --list" for available build arguments.
enable_nacl = false
is_asan = true
is_debug = false
clang_base_path = getenv("HOME") + "/build"
clang_use_chrome_plugins = false
is_debug = false
symbol_level = 1
is_component_build = true
use_lld = false
use_sanitizer_coverage = true
sanitizer_coverage_flags = "trace-pc-guard"
```

FIGURE 3.6: Chromium Build Arguments

Our scripts to link up different components in series, build coverage database, score samples and launch refinement are all written in Python and Bash language. Functionality and usage of part of these scripts are described as follows in their running sequence:

- **generator.py**: Domato's generator script to generate DOM test cases. We make a few modification that enables it to use a specified template given by user for generation. Its usage is as:

```
python generator.py --output_dir <output directory>
--no_of_files <number of output files> --template <specified
template path>
```

- **coverage.sh**: Feed a test case to the fuzzing target and save the coverage data to a specified directory through command:

```
ASAN_OPTIONS="coverage=1:detect_odr_violation=0:
coverage_dir =<output directory>" timeout <time>
~/chromium/src/out/asan/chrome <test case> & >log
```

In `ASAN_OPTIONS`, with setting `coverage=1`, SanitizerCoverage will write data of covered edges into one .sancov file for every executed binary/DSO. In the meanwhile, `detect_odr_violation` must be set to zero, otherwise, ASAN will halt the target process with reporting One Definition Rule(ODR) violation which is meaningless for our fuzzing cases. And by some initial checking, we find most test cases can be completely loaded within 40 seconds so we set the timeout duration as 40s.

Finally, the executing result outputted by ASAN is extracted to a log file for crash identification. Here the identification is implemented in a naive way, which is to check whether an error message arises in the log, through string matching by an if-statement:

```
if grep -q "ERROR: AddressSanitizer:" "log";then
```

Once the above statement is true, the log, as well as the current test case, will be saved to a specified crash archive for further manual analysis.

- **cov_collect.py**: Collect and merge coverage data obtained by coverage.sh into a json file for every test case. Also, count the number of hits for every covered edge, save all the edge-counter pairs in our database:

```
python cov_collect.py <input coverage directory> <output
coverage directory> <output database path>
```

- **scoring.py**: Look through the coverage database with a threshold value. Output a set of rarely-hit edges whose number of hit is lower than the threshold, as well as all interesting test cases with their IDs and rarely-hit edges hit by them in a json file:

```
python scoring.py <threshold> <input database path> <output
rarely-hit edges path> <output interesting samples path>
```

- **reproduce.py**: Check whether an interesting test case performs in a reproducible way:

```
python reproduce.py <number of check rounds> <input
interesting samples path> <output reproducible samples path>
```

  Here `<input interesting samples path>` is the path to the json file obtained by scoring.py which includes ID and set of rarely-hit edges for every interesting test case. The script calls coverage.sh to re-execute these test cases for times given in `<number of check rounds>`. Its output is also a json file whose path is specified in `<output reproducible samples path>` storing IDs and rarely-hit edges of test cases that are marked as reproducible.

- **Minimize.py**: Minimize the reproducible test cases by removing as many redundant statements as possible:

```
python Minimize.py <input reproducible samples path> <output
minimized samples path>
```

  This script extracts IDs and rarely-hit edges of samples from json file in `<input reproducible samples path>`, and uses such data as inputs to Algorithm 5 to minimize every sample. The finally minimized samples are saved in `<output minimized samples path>`.

After a round of above operations, we get a set of minimized templates and use them to generate new test cases in the next round.

For evaluation, we run two rounds of our approach:

In the first round, we generate a massive amount of test cases by using the default template provided by Domato, then feed them to the instrumented target program

to compute edge coverage, build the database, find interesting samples and finally get a set of interesting templates.

In the second round, we implement a contrast experiment: generate two sets of test cases of the same amount, one is based on the default template, the other one is based on the new interesting templates obtained in the first round. Then we compute and compare the edge coverage of these two sets to evaluate the performance of our approach in coverage improvement.

Generation, execution and refinement of test cases are implemented with multiple virtual machines in VirtualBox 5.2.20 [67]. All the virtual machines are configured with the same settings: 64-bit Ubuntu 18.04 as the operating system, with 8GB RAM and 2 CPUs. Every virtual machine runs independently but read/write data from/into a shared directory. Analysis of coverage data and scoring of samples are performed in the host machine. Coverage database and scoring results are also saved into the shared directory for accessing by these virtual machines.

Data	Amount
Test cases	120,000
Total covered edges	237,493
Maximum # of edges in one sample	185,366
Minimum # of edges in one sample	76,076
Average # of edges in one sample	176,107
Maximum hit times of one edge	120,000
Minimum hit times of one edge	1

TABLE 3.1: Basic Results of Execution in Round 1

We have generated 120,000 test cases in the first round. As shown in Table 3.1, there are in total 237,493 edges covered in these test cases. The maximum amount of edges triggered in one single test case is 185,366, and the minimum amount is 76,076. By average, every test case triggers 176,107 edges. And there exist edges that triggered by all of 120,000 test case, as well as edges that triggered only once.

Then we think about the choice of the threshold used to determine "rarely-hit" in scoring. We have tried three threshold values: 1, 10 and 100. The results are

Threshold	1	10	100
Rare edge amount	4,340	14,831	24,909
Interesting samples amount	72	611	5,781
Ratio of interesting	0.06%	0.51%	4.82%

TABLE 3.2: Threshold Related Result

described in Table 3.2. Since there are around 0.06% test cases able to hit the rarest edge, we just select 1 as the threshold. So in the first round, we get 72 test cases marked as interesting.

Number of interesting test cases	72
Number of reproducible	68
Average size before minimization	494.82KB
Average size after minimization	5.63KB

TABLE 3.3: Results of Refinement in Round 1

The results of refinement for these interesting test cases is shown in Table 3.3. We find that 68 of 72 test cases are reproducible upon 10 checking rounds, and our minimization approach has successfully reduced 98.86% of their average size.

We get 68 minimized interesting test cases to be used as templates in our second round. To evaluate the code coverage improvement, we use these new templates to generate sample set, marked as set A, use the original default template to generate another sample set, marked as set B. We compare the code coverage of 5 pairs of set A and B, where the sample amount of two sets are equal in each pair. And for set A, every new template generates an equal amount of new samples, meaning the set size is always a multiple of 68.

Set size	3,400	6,800	13,600	34,000	68,000
Set A	214,326	219,463	231,489	242,188	244,904
Set B	206,806	210,248	216,628	228,561	231,831
Outperformance	3.64%	4.38%	6.86%	5.97%	5.64%

TABLE 3.4: Code Coverage Comparison

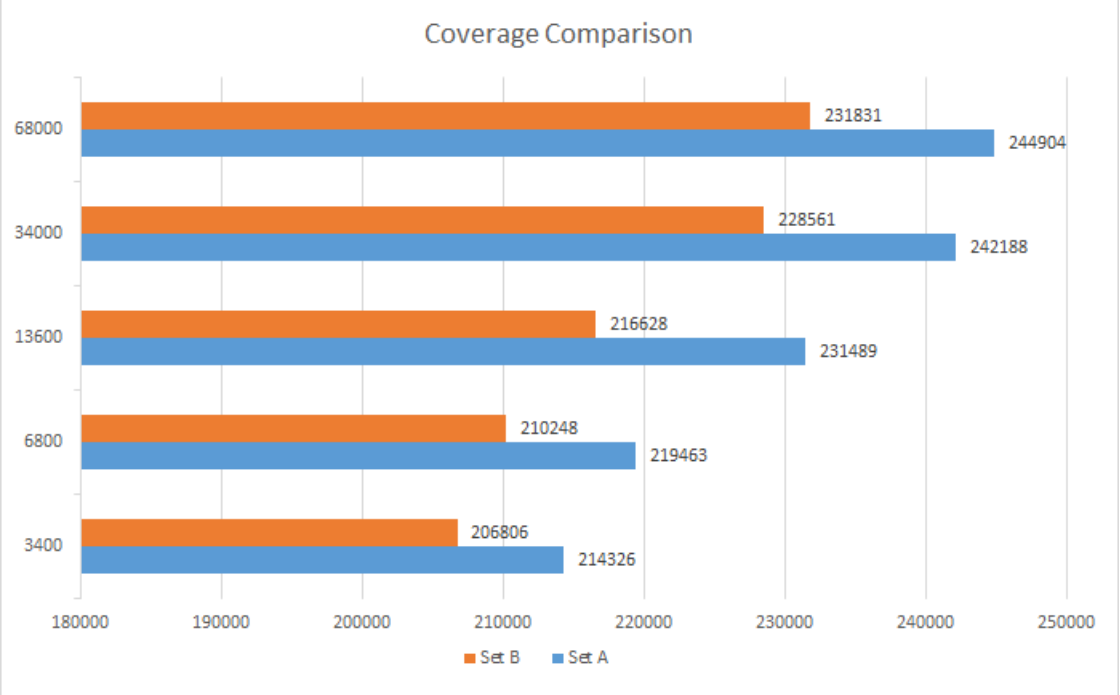


FIGURE 3.7: Code Coverage Comparison

Table 3.4 and Figure 3.7 give the results of our contrast experiments. We see that set A outperforms set B in all the 5 different set sizes. With set size increasing from 3,400 to 13,600, the outperformance percentage grows, then appears slightly drop with size from 34,000 to 68,000. That is expected because along with the growth of test cases amount, set B can hit some rarely-hit edges which are hit much more earlier by set A. However, set A is able to get in total 242,188 edges covered with only 34,000 samples, even higher than 237,493 edges covered by 120,000 samples in our first round. It demonstrates that our approach has successfully increased the edge coverage for generative fuzzing on Chromium DOM.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

We have presented two works about improving the efficiency of fuzzing, one of the most widely used techniques for vulnerability detection.

In the first work in Chapter 2, we analyzed the attack surface for Linux Kernel fuzzing. We collected CVEs related to Linux Kernel from year 1999 to 2018, and mapped each CVE to one or more Linux Kernel components upon its affected files. We analyzed the mapping results and summarized CVEs distribution on the kernel component. Finally, we drew a vulnerable level map of Linux Kernel components which could identify valuable parts for attack thus benefit the efficiency of fuzzing.

In the second work in Chapter 3, with inspiration from FairFuzz proposed by Lemieux, C. & Sen, K.[15], we presented a rarely-hit edge targeted approach to improve edge coverage for generative fuzzing on chromium DOM. We instrumented the target program for crash detection and coverage computation propose, then counted the number of hits for each covered edge and used edges that were rarely hit as metric to score interesting samples. We refined these interesting samples into templates to generate new samples in the next round of fuzzing. Through comparing the performance of samples generated by new and default templates, our approach demonstrated an obvious improvement on fuzzing code coverage.

4.2 Future Work

Both the two works have limitations that could be further improved:

For the attack surface analysis, our current CVE-Component mapping doesn't consider the different versions of Linux Kernel, which could be distinguished more specifically in the future. In addition, a category in more detail leads to more accurate identification of the location of potentially vulnerable parts. Our current work is based on functional component level, and in future we could research on more subtle levels such as function or basic block level.

For the guided generative fuzzing, we need to build a framework by integrating all the steps and scripts, aiming at implementing a fully automatic and continuous fuzzing with multiple rounds. To ensure the deterministic performance of our approach on code coverage improvement, experiments with a larger amount of test cases as well as generation rounds are also needed. Moreover, we could improve the efficiency of coverage computation by manually defining the functions instrumented in the target program.

Bibliography

[1] CVSS score distribution from 2016 to 2018. Retrieved from `https://www.cvedetails.com/cvss-score-charts.php`.

[2] Cyber crime costs global economy $445 billion a year: report. Retrieved from `https://www.reuters.com/article/us-cybersecurity-mcafee-csis/idUSKBN0EK0SV20140609`.

[3] Wannacry ransomware attack. Retrieved from `https://en.wikipedia.org/wiki/WannaCry_ransomware_attack`.

[4] International Telecommunications Union (ITU). *ITU-TX.1205: series X: data networks, open system communications and security: telecommunication security: overview of cybersecurity.* 2008.

[5] Alexander Klimburg. *National cyber security framework manual.* NATO Cooperative Cyber Defense Center of Excellence, 2012.

[6] Joint Task Force Transformation Initiative Interagency Working Group et al. NIST special publication 800-53 revision 4-security and privacy controls for federal information systems and organizations. *National Institute of Standards and Technology*, 2013.

[7] Browse vulnerabilities by date. Retrieved from `https://www.cvedetails.com/browse-by-date.php`.

[8] SQL. Retrieved from `https://en.wikipedia.org/wiki/SQL`.

[9] Alexander Ivanov Sotirov. *Automatic vulnerability detection using static source code analysis.* PhD thesis, Citeseer, 2005.

[10] Jan Van Lunteren. High-performance pattern-matching for intrusion detection. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–13. Citeseer, 2006.

[11] Sarang Dharmapurikar and John W Lockwood. Fast and scalable pattern matching for network intrusion detection systems. *IEEE Journal on Selected Areas in communications*, 24(10):1781–1792, 2006.

[12] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[13] Zhemin Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *2012 Third World Congress on Software Engineering*, pages 101–104. IEEE, 2012.

[14] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[15] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485. ACM, 2018.

[16] Brian S Pak. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *School of Computer Science Carnegie Mellon University*, 2012.

[17] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[18] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. *ACM Transactions on Information and System Security (TISSEC)*, 14(2):15, 2011.

[19] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[20] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[21] Announcing Project Springfield. Allison linn. *Microsoft AI Blog*, 2016.

[22] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. Announcing oss-fuzz: Continuous fuzzing for open source software. *Google Testing Blog*, 2016.

[23] Interactive disassembler. Retrieved from `https://www.hex-rays.com/products/ida/`.

[24] gdb: the gnu project debugger. Retrieved from `https://www.gnu.org/software/gdb/`.

[25] Domato. Retrieved from `https://github.com/googleprojectzero/domato`.

[26] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[27] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.

[28] American fuzzy lop. Retrieved from `http://lcamtuf.coredump.cx/afl/`.

[29] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[30] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 37–48. ACM, 2014.

[31] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*, pages 330–347. Springer, 2015.

[32] K Serebryany. libfuzzer a library for coverage-guided fuzz testing. *LLVM project*, 2015.

[33] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[34] Ulf Kargén and Nahid Shahmehri. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 782–792. ACM, 2015.

[35] Pratyusa K Manadhata and Jeannette M Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2010.

[36] J Bird and J Manico. Owasp attack surface analysis cheat sheet. *Open Web Application Security Project*, 2015.

[37] Michael Howard, Jon Pincus, and Jeannette M Wing. Measuring relative attack surfaces. In *Computer security in the 21st century*, pages 109–137. Springer, 2005.

[38] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 315–317. ACM, 2008.

[39] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. *arXiv preprint arXiv:1901.11479*, 2019.

[40] Interview: Linus Torvalds. Retrieved from `https://www.linuxjournal.com/article/3655`.

[41] Operating system family/linux. Retrieved from https://www.top500.org/statistics/details/osfam/1.

[42] Trinity: A linux system call fuzz tester. Retrieved from http://codemonkey.org.uk/projects/trinity/.

[43] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2017.

[44] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for {OS} kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[45] The linux kernel has grown by 225,000 lines of code this year, . Retrieved from https://linux.slashdot.org/story/18/09/16/172217/.

[46] Beautiful soup documentation. Retrieved from https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[47] Parsing xml and html with lxml. Retrieved from https://lxml.de/parsing.html.

[48] National vulnerability database. Retrieved from https://nvd.nist.gov/.

[49] Cvedetails: The ultimate security vulnerability datasource, . Retrieved from https://www.cvedetails.com.

[50] CVE-2018-1857, . Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2018-1857.

[51] CVE-2018-1786, . Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2018-1786.

[52] CVE-2018-1834, . Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2018-1834.

[53] Linux kernel: Vulnerability statistics. Retrieved from https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[54] Kernel.org git repositories. Retrieved from https://git.kernel.org/.

[55] M Tim Jones. Anatomy of the linux kernel: History and architectural decomposition. *IBM developerWorks*, 2007.

[56] Interactive map of Linux Kernel, . Retrieved from http://www.makelinux.net/kernel_map/.

[57] Bootlin. Retrieved from https://bootlin.com/.

[58] Honggfuzz. Retrieved from http://code.google.com/p/honggfuzz.

[59] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637. ACM, 2017.

[60] Jesse Ruderman. Introducing jsfunfuzz. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz, 2007.

[61] The chromium projects. Retrieved from https://www.chromium.org.

[62] Clang doucumentation: Sanitizercoverage. Retrieved from https://clang.llvm.org/docs/SanitizerCoverage.html.

[63] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[64] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74. ACM, 2007.

[65] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5, 2008.

[66] Clang 8 documentation. Retrieved from `https://releases.llvm.org/8.0.0/tools/clang/docs/ReleaseNotes.html`.

[67] Jon Watson. Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1, 2008.