國立交通大學

資訊科學與工程研究所

碩 士 論 文

具目標認知符號執行模糊測試框架

A Target-Aware Symbolic Execution Framework for Fuzz Testing

研 究 生：鍾　翔

指導教授：黃世昆　教授

中 華 民 國 一 百 零 三 年 五 月

具目標認知符號執行軟體測試框架

A Target-Aware Symbolic Execution Framework for Fuzz Testing

研 究 生：鍾　翔　　Student：Hsiang Chung

指導教授：黃世昆　　Advisor：Dr. Shih-Kun Huang

國 立 交 通 大 學
資訊科學與工程研究所
碩 士 論 文

A Thesis
Submitted to College of Computer Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in

Computer Science

June 2014

Hsinchu, Taiwan, Republic of China

具目標認知符號執行模糊測試框架

學生: 鍾　翔　　　　　　　　　　　　　　指導教授: 黃世昆

國 立 交 通 大 學　　資 訊 科 學 與 工 程 研 究 所　　碩 士 班

摘要

軟體設計不良所產生的漏洞，例如 buffer overflows、integer overflows、uncontrolled format strings 和 command injections 等，這些問題常被駭客操作使用、入侵使用者個人電腦或伺服器。Windows 和 Linux 上的應用程式，或作業系統本身不時發布安全性更新就是為了修補這樣的問題。

為了減少軟體的漏洞，有許多測試方法被提出來，其中最常使用的是模糊測試（fuzz testing）。但傳統的模糊測試必須執行到程式出現例外情況（如失控）才能發現該問題，導致覆蓋率不足時無法發現受測程式的漏洞，忽略可能存在的安全威脅。

本篇論文提出使用 *S2E* 以 symbolic execution 為基礎的軟體測試架構，能在程式正常執行到某些自訂的敏感函式，例如 *malloc*、*strcpy* 和 *printf* 時，自動判斷此程式執行路徑在此位置是否可能造成安全性的威脅，若是，則進一步產生 exploit 的概念驗證（proof of concept），以及相對應的數學限制式。

我們運用此方法成功且有效地產生許多在 CVE 網站公開的漏洞，並能協助開發者迅速找到問題所在，提升維護軟體品質的效率。

A Target-Aware Symbolic Execution Framework for Fuzz Testing

Student: Hsiang Chung　　　　Advisor：Dr. Shih-Kun Huang

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Vulnerabilities caused by implementation bugs, such as buffer overflows, integer overflows, uncontrolled format strings, and command injections, are often exploited by hackers to intrude users' personal computer or servers. In order to reduce software bugs, many testing techniques are proposed. The most frequently used technique is fuzz testing. However, traditional fuzzers can only find bugs when program exceptions, especially crashes, raised. That means some security threats may pass these tests due to the insufficient code coverage.

In this thesis, we introduce a software testing framework based on symbolic execution using *S2E,* a whole system symbolic execution engine. When a program executes some pre-defined sensitive functions, such as *malloc*, *strcpy* or *printf*, our framework will initiate a triage process. It will determine whether any related security vulnerabilities would possibly occur in these functions automatically. If the answer is yes, a proof-of-concept exploit and its corresponding math constraints will be generated.

We successfully and efficiently reproduce some CVE vulnerabilities, which means developers could locate bugs faster, and improve the efficiency of software quality maintenance.

誌謝

首先我要感謝我的指導教授黃世昆老師，黃老師從我進實驗室開始就對我十分地照顧，不論在做研究、待人處事、還是生活態度上都能給我很大的幫助，能當老師的學生是我大學生涯最幸運的事了。在大學六年的生活中，各科教授、和我吃了六年導聚的林正中導師及美麗的系辦姊姊們也是我成長的推手。

我十分感謝我的家人，他們總是默默地支持我，相信我，對我做的任何決定都給予相當的肯定與尊重。住在新竹的姑姑和姑爹，每當我有需要的時候總是能去尋求他們的幫助，在交大生活的這段日子裡他們幫上我許多忙，非常感激他們。

SQLab 的同學們當然不能漏掉，尤其是翰霖學長，每次都能不厭其煩地解決我難搞的問題。還有自我創立網路福利社就很照顧我的俊憲學長，在實驗室和系計中都特別照顧我的 Jeff、linpc、韋翔、偉明學長，陪我打暗黑打到三四點的俊維學長，幽默風趣的基傑學長和在研究之路給我很多中肯建議的肇鈞和銘祥學長，常常被我煩的劉歡、貓咪和 Lance，相當厲害的學弟 KY、達達、紹文和 Vince。

大學的同學們也是我努力動力的來源，感謝一起和我對抗 DVD 的 csSula 成員大兜、Kevin、大類、惟中和文莉，臭味相投的好友牛人兼 skip 戰友邱渣和丁牛，以及很照顧我的其他東京熱大伊姆拉拉、大 RR 凱吉和阿禧，在這邊先預祝他們博士班畢業快樂。其他還有陪我走過大風大浪的知已欣欣，資工系計中非常照顧我的團長大人、hhyou、振全、巨龍、liuyh、柏明、乙澔、伯羽、游傑、changlp、ywang、寬寬、孟儒、Jalen、ylin、天天、阿佩、易侁、笨雞、嘉駿、Mars，相處數年的同學煥博、包子、賊魚、貓耳、Eric、賴聖帝君、詩庭、子瑄、君君、溫溫，多次幫我校稿碩論和擔任司機的室友陳揚、柏榕和沒事會請我喝酒的定宇。

謝謝你們以及其他因為寫不下沒列在裡面，陪伴我、幫助我的朋友及師長，沒有你們就沒有今日的鍾翔。

Table of Contents

摘要 i
Abstract ii
誌謝 iii
Table of Contents iv
Table of Tables v
Table of Figures vi
CHAPTER 1 INTRODUCTION 1
CHAPTER 2 OVERVIEW 6
A. Symbolic Execution 6
B. Symbolic Execution Optimization 8
C. *S2E* 9
D. Vulnerable Situations 11
E. DLL Injection 13
CHAPTER 3 Method 14
A. Test Case Acquisition 15
B. Target Searching 16
C. Proof-of-Concept Generation 17
D. Verification 20
E. Example: *XMail-1.21* 20
F. Implementation 23
CHAPTER 4 EVALUATION 27
A. Experiment Setup 28
B. Hot Bytes Identification 29
C. Adaptive-Input Technique Evaluation 31
D. Null-Constraint Technique Evaluation 32
E. Fuzzing Results 34
F. Case Studies 35
CHAPTER 5 CONCLUSION 39
References 42

Table of Tables

TABLE 1: COMPARISON OF TRADITIONAL FUZZERS WITH *CRAXFUZZER* 23
TABLE 2: LIST OF DISCOVERED SENSITIVE FUNCTIONS 24
TABLE 3: LIST OF TESTING APPLICATIONS 27
TABLE 4: HOT BYTES IDENTIFICATION RESULTS 28
TABLE 5: ADAPTIVE-INPUT TECHNIQUE RESULTS 32
TABLE 6: NULL-CONSTRAINT TECHNIQUE RESULTS 33
TABLE 7: CRAXFUZZER FUZZING RESULTS 34

Table of Figures

FIGURE 1: SYMBOLIC EXECUTION PATH EXPLOSION EXAMPLE 3
FIGURE 2: SYMBOLIC EXECUTION TESTING PROGRAM 7
FIGURE 3: *S2E* ARCHITECTURE I 10
FIGURE 4: *S2E* ARCHITECTURE II 10
FIGURE 5: TRADITIONAL FUZZER ARCHITECTURE OVERVIEW 15
FIGURE 6: TARGET-AWARE SYMBOLIC EXECUTION FRAMEWORK OVERVIEW..............15
FIGURE 7: TARGET SEARCHING OVERVIEW 17
FIGURE 8: TRANSFORMATION FUNCTION EXAMPLE 17
FIGURE 9: TRANSFORMATION RELATIONS IN PROGRAM EXECUTION 17
FIGURE 10: SAMPLE EMAIL WITH ENVELOPES FOR *XMAIL* 21
FIGURE 11: FUNCTION *ADDRESSFROMATPTR* IN *SENDMAIL.CPP* IN *XMAIL* 21
FIGURE 12: *STRNCPY* ARGUMENT EXPRESSIONS OF *INPUT* IN *XMAIL* 22
FIGURE 13: *CRAXFUZZER* ARCHITECTURE..23
FIGURE 14: *CRAXFUZZER* COMPONENT RELATIONS 26
FIGURE 15: HOT BYTES OF TIFF FILE I 30
FIGURE 16: HOT BYTES OF TIFF FILE II 31
FIGURE 17: HOT BYTES OF FILE USING *FILETYPE.VIM*..31
FIGURE 18: PARTS OF *SUDO* CODE 36
FIGURE 19: PARTS OF *TIFF* CODE...37
FIGURE 20: REPORT OF *MALLOC* OF *TIFF* EXECUTION 37
FIGURE 21: VULNERABILITY NOT RELATED TO FUNCTIONS EXAMPLE 41

CHAPTER 1 INTRODUCTION

Due to the rapid development of information technology and Internet, people can easily install software and download files. However, defective programs may contain security vulnerabilities such as buffer overflows, integer overflows, uncontrolled format strings, and command injections. An attacker can exploit these vulnerabilities by feeding properly designed input files and take control of the victim's systems. Security patches are issued for applications on Windows, Linux, or the operation system itself every day to solve these problems [2]. Thus, information security has become an important issue for normal users and enterprises.

In order to reduce software bugs, many testing techniques are proposed. The one most frequently used is fuzz testing, or called fuzzing [26], which has been proven successful in finding bugs and security vulnerabilities in large software applications. The idea behind fuzzing is very simple. First, we generate inputs fed to the program to be tested. If exceptions are raised, which often result in a crash, a potential security issue is detected. A great number of severe software vulnerabilities have been revealed by fuzzing techniques and related researches [40]. For example, the *CVE (Common Vulnerabilities and Exposures)* website[1] lists massive vulnerabilities, and some are marked or found to be fuzzed easily.

[1] http://cve.mitre.org/

TaintScope [42] used its checksum-aware technique to fix checksum fields on the program input header and successfully identified many known and previously unknown program vulnerabilities. Popular fuzzing tools include *zzuf* [17], a dumb fuzzer which generates inputs randomly, and *Peach* [12], a format-aware fuzzing platform that can model input data structures. *M. Woo, et al* [45] integrates many existing scheduling algorithms and has good efficiency.

Since most applications are with unlimited input space, traditional fuzz testing tools have an inherent limitation of low code coverage. This means that serious security bugs may be missed because the code in which they exist is not even executed. Many techniques are proposed to improve code coverage, one of which is symbolic execution [19, 34], a constraint solving based system. It substitutes symbolic values for program input bytes, gathering path and input constraints while encountering a branch. By solving these gathered symbolic constraints, we can generate new inputs for almost all the running program paths and thus a good code coverage is reached. In recent years, researchers have found many new security vulnerabilities by symbolic-execution-based fuzzers. *SAGE* [15] applied their generational search algorithm to find many bugs in a variety of Windows applications. *EXE* [28] has found many critical bugs from Linux ports, including image viewers and media players. *BitBlaze* [27, 38] and *S2E* [8, 9] are two large-scale symbolic testing platforms. *BitBlaze* uses *TEMU* [38], while *S2E* chooses *KLEE* [5] as the symbolic execution engine. We build our framework based on *S2E*.

```
input (int  x)
 while( x > 0 )
 { ... }
```

Figure 1: Symbolic Execution Path Explosion Example

Unfortunately, path explosion is a big problem of symbolic execution. Although symbolic execution could cover most of the program paths, the number of feasible paths in a program grows exponentially with an increase of program size or with just a loop iteration [1]. **Figure 1** shows an example of loop path explosion. If the input variable *x* is symbolic, symbolic executor will try to list all the possibility that satisfied “*x > 0*” and run the corresponding paths. Most parts of this action are redundant.

To speed up symbolic execution, some solutions are proposed. One common way is concolic testing [35], which combines concrete and symbolic execution. Concolic execution gives the program an initial input, so it can follow the input deeper in the code. *CUTE* [36] is an instance of concolic execution. Alternatively, it is another possible way to improve path selection algorithms. *Ma, K.-K., et al.* [23] proposed shortest-distance and call-chain-backward as two heuristics for path-finding, while *STrigger* [22] used a weighted search algorithm based on the control flow graph (CFG). Another approach is to control symbolic path space by selecting input bytes [47]. *Spat* [46] applies their partial symbolic execution that tracks only a prefix of the input data, which is related to this approach. Other researchers choose to shorten execution time by paralleling running [39] or by cutting paths to be explored into pieces [7].

We have discussed lots of software testing techniques. Please note that most of them are performing passive testing, which means that they just generate a new input, testing if it crashes, and then generate a new one again. There is no threating target to be searched or a guideline to be followed. *TaintScope* [42, 43] uses taint-like analysis [31] to mark every input bytes and see if some security-sensitive points could affect these input bytes. Tainted input bytes are called *hot bytes*, which means they can directly influence the context of security-sensitive operations. Since *TaintScope* knows only which bytes are tainted, it has to run symbolic execution additionally to generate crash inputs. Recently, many symbolic execution tools are proposed to deal with some vulnerabilities. *Splat* [46] defines a buffer overflow situation and *Catchconv* [29] defines an integer conversion error situation, while *IntScope* [44] defines an integer overflow situation and *Saxena, P., et al.* [33] defines a loop-extended situation. However, they merely focus on one specific condition, and it is hard to generalize the problems. *BuzzFuzz* [14] uses a directed dynamic taint-based white-box fuzzing technique which requires to instrument an application's source code. *Caselden, D., et al.* [7], *McCamant, S., et al.* [25] and *STrigger* [22] introduced vulnerability-condition-based, or trigger-condition-based test case generation methods respectively, but no significant results have been revealed.

In this thesis, we proposed a target-aware symbolic execution framework for fuzz testing. Our work can find bugs caused by specified library functions and prove it in a short time. We generate a proof-of-concept exploit instead of only a crash input. Unlike traditional fuzzers to generate crash input, we think crashes are not necessary if we have enough information to produce exploits. We further provide tips to reduce software testing and symbolic execution time. The primary contributions to our work are described as following.

- We introduce a technique to hook target functions in standard libraries, such as *malloc*, *strcpy* and *printf*. We define and generalize these sensitive points and test whether there are possible vulnerabilities or not, and then generate a prove-of-concept exploit by solving constraints further.
- We introduce a method to identify hot bytes of files and obtain their relations to headers.
- We introduce a whole-system fuzzing framework that can analyze not only applications but also libraries, drivers, or operating system (OS) itself without source codes.
- We introduce techniques to speed up symbolic execution by dropping unnecessary path constraints or using adaptive symbolic inputs.
- We evaluate the effectiveness of our method by applying our methods on existing CVE vulnerable software. We also provide case studies to show the profit of our work.

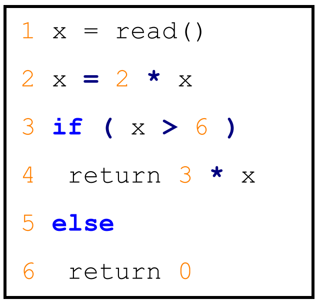
CHAPTER 2 OVERVIEW

In this chapter, we give a technical overview of symbolic execution, its optimization and the symbolic engine we choose. We also introduce the method to hook functions and provide some vulnerable situations as our fuzzing targets.

A. Symbolic Execution

Symbolic execution is a dynamic software analysis technique that analyzes a program path-by-path, which is an advantage over analyzing a program input-by-input such as traditional fuzz testing methods. If two inputs take the same path through the program, the testing by means of the path will save more time than that by means of the inputs. When exploring paths, symbolic executor also gathers corresponding constraints. We therefore know how to get to this path, and are able to modify them to fit our requirement, a crash situation for example. In our approach, we use symbolic execution to search for hooked functions and record constraints between program inputs and function arguments.

Symbolic execution uses symbolic values instead of concrete data on program inputs. An interpreter executes, assuming values rather than obtaining them from actual inputs, unlike normal program executions. In this way, it learns relations in terms of those symbols for expressions and variables when arriving the target location, and the path constraints for reaching this position is also learned [34]. We can solve these constraints of each conditional branch by a decision procedure, or a constraint solver, such as *STP* [13] or *Z3* [11]. If a solution exists, we could find a new program path.



```
1 x = read()
2 x = 2 * x
3 if ( x > 6 )
4  return 3 * x
5 else
6  return 0
```

Figure 2: Symbolic Execution Testing Program

Consider the program shows in **Figure 2**, which reads a value and returns a value of six times *x* if the input *x* is greater than 3, and returns 0 else. When a symbolic executor runs this program, it does not have a concrete value for the input value, i.e., the result read from **line 1**. Alternatively, the executor assigns this program a symbol *s* to the concrete value. Then statement "*x = read()*" assigns *s* to program variable *x*. And in **line 2**, the statement "*x = 2 * x*" assigns *2 * s* to *x*. The next statement in **line 3** has two conditions: the true branch and the false branch, which depend on our input value *s*. The executor associates the constraint "2 * s > 6" with the true branch, which means that the program returns *3 * x* if and only if "*2 * s > 6*" is true. And it combines the constraints "*NOT (2 * s > 6)*" with the false branch, which negates the true branch as a new path and make the program return 0. Note that the returned value "*3 * x*" in **line 4** was substituted by symbolic value"*3 * 2 * s*", which is known as the return argument expression, and that "*2 * s > 6*" and "*NOT (2 * s > 6)*" are two different path constraints. Assume we want "return *3 * x*" in **line 4** to be executed, we can use a constraint solver mentioned above to determine a value to make "*2 * s == 6*" true. If we want the program to return 24 further, we should also make expression "*3 * 2 * s*" to be 24, to which a constraint "*3 * 2 * s == 24*" should be added. Combining two constraints, we will get “*2 * s > 6*” and “*3 * 2 * s == 24*”. Solve the both and we will get a value of input x to force this program to return 24, the result we want.

B. Symbolic Execution Optimization

We use several techniques to optimize our symbolic execution. Some are mentioned in the **Introduction** above and some will be proposed as follows.

I. Adaptive-Input Symbolic Execution

Because program inputs which we make symbolic are often very large. For example, a “*.doc*” document is an input file for *Microsoft Word*, and those documents may be millions of bytes in size. In such a case, running a complete symbolic execution may take hours or days, which is unacceptable.

To improve this situation, an adaptive input based method, which symbolizes only parts of input space, has been proposed and verified [47]. However, which part of the input is more important is a question. We believe that the header part of an input may gain more benefits in term of execution efficiency due to the influence of important data structures. We also found that it is more efficient to split an input into segments than to test a whole input in the concept of divide-and-conquer. We evaluate these in **Section 4-C**.

II. Concolic Execution

The main idea of concolic execution is running the testing program symbolically with a concrete input. It can follow this input going deeper into the code. In some cases, we don’t want to be blocked by any integrity checking functions and exit too early. A well-formed input, which usually produces a good code coverage, could help us to stay in the deep code by extending this program trace. The efficiency of this technique has been proved by those tools such as *KLEE* [5].

III. Null-Constraint Single-Path Concolic Execution

Sometimes, one well-formed input could provide enough information. We want to avoid our concolic execution being lost in the code or stuck in a loop, so we disable the forking process for producing new branches to focus on one program trace. We name this skill single-path concolic execution.

In a single-path concolic execution, gathering path constraints is not necessary anymore because only one concrete path will be executed. Assuming that we want only symbolic expressions or a taint-like [31] functionality, these constraints could be dropped. If we want to reduce testing input space, this technique is also a good choice. We call this null-constraint single-path concolic execution. The good efficiency of null-constraint will be shown in **Section 4-D**.

C. *S2E*

S2E [8] is a whole-system symbolic-execution-based automated path explorer with modular path analyzers. The explorer expands all the paths in which we are interested, and the analyzer looks for bugs of each such path or simply collects information.

Figure 3 and **Figure 4** shows the *S2E* architecture in high-level and mid-level respectively. The prototype of this platform reuses parts of the *QEMU* virtual machine [4], the *KLEE* symbolic execution engine [5], and *LLVM* tool chain [20]. It can execute any gest OSs that runs on an *x86* or ARM CPU.

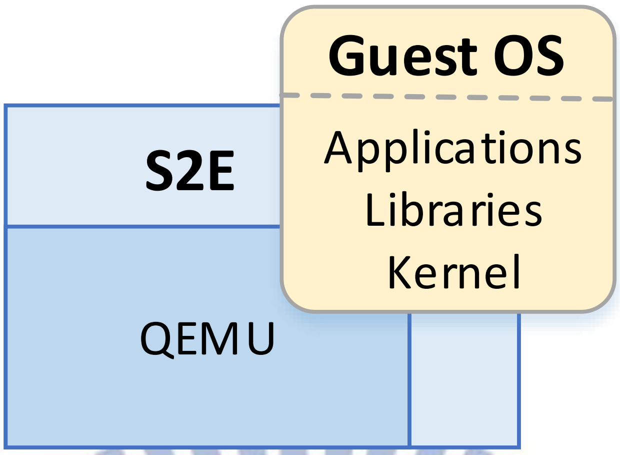


Figure 3: *S2E* Architecture I

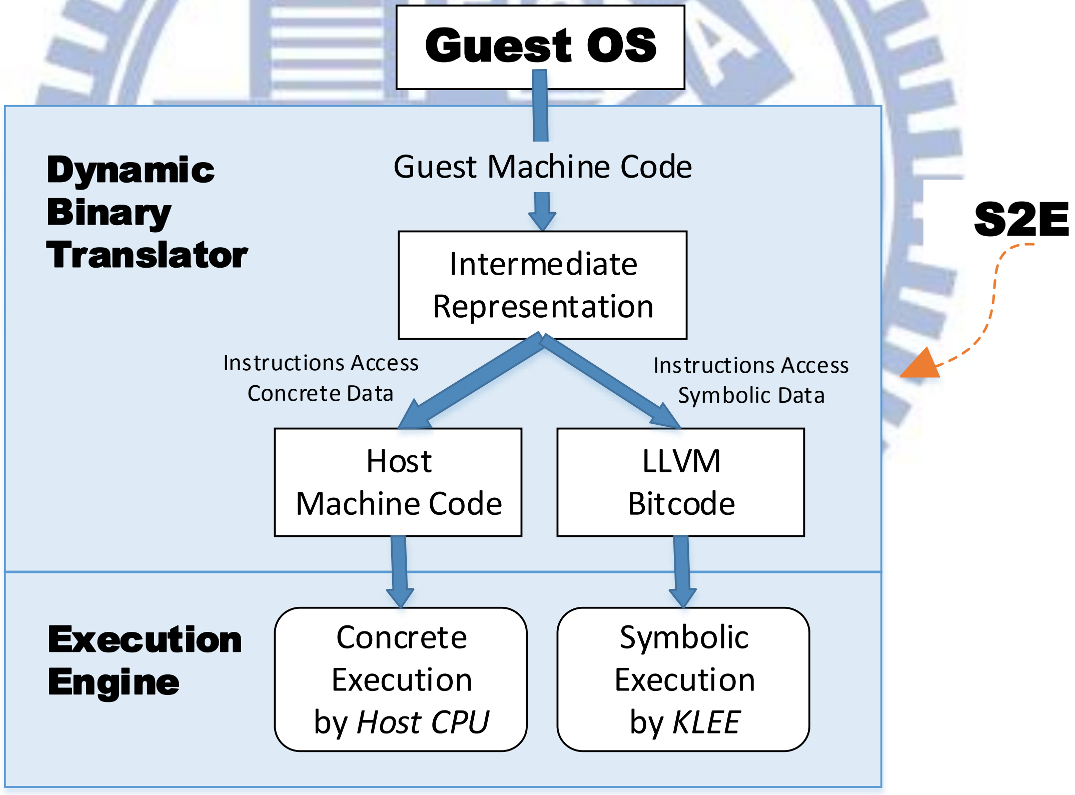

Guest OS
Dynamic Binary Translator
Guest Machine Code
Intermediate Representation
Instructions Access Concrete Data
Instructions Access Symbolic Data
Host Machine Code
LLVM Bitcode
Execution Engine
Concrete Execution by Host CPU
Symbolic Execution by KLEE
S2E

Figure 4: *S2E* Architecture II

S2E explores paths by running the target system image and selectively executing small parts of it symbolically. Depending on which segment of code we desired, the corresponding system's machine instructions are dynamically translated within the virtual machine into an intermediate representation suitable for symbolic execution, while the rest are translated to host instruction set as normal binary translation. Because all of the symbolic and concrete executions are done outside the guest machine, a full system (OSs, libraries, applications, etc.) testing for the guest system could be applied.

S2E is easy to use. It modified *QEMU*'s dynamic binary translator (DBT) to translate the instructions that depend on symbolic data to *LLVM*, and dispatch them to *KLEE*. In this way, users can test any binary codes that run in the guest OS without any source. Due to the open source agreement of *QEMU*, developers can easily modified *S2E* to fit their requirement. Thus we use *S2E* as our core engine.

D. Vulnerable Situations

There are hundreds of vulnerabilities. The *Top 25 Most Dangerous Software Errors* [24] lists the most widespread and critical errors that can lead to serious vulnerabilities in software. We pick up four cases which are often seen in *C* programs as our fuzzing target situations.

I. Buffer Overflow

Buffer overflow is an important and persistent security problem and counts for approximately half of all security vulnerabilities in recent years [10]. This problem occurs when more data are written to a buffer than it can hold. The excessive data is written to the adjacent memory, overwriting the contents including returned addresses in the stack memory. Many memory-based functions in the standard library are easy to cause buffer overflows. We will discuss this later.

II. Integer Overflow

Integer overflow is a generic name of integer errors such as overflow, underflow, and signed/unsigned conversion errors. *CVE-2002-0639* about *OpenSSH* and *CVE-2010-2753* about *Firefox* are two serous integer vulnerabilities. Many integer overflow vulnerabilities are closely related to memory allocation functions [44]. If an integer input is used to restrict a memory manipulation without exhaustive checks, memory violation errors could occur. Take *malloc* as an example. If the size argument overflows, the operating system will allocate less memory space than the program wants, than a heap overflow would happen.

III. Uncontrolled Format String

A format string is an *ASCII* string that contains text and format parameters. When a format function, *printf* for example, evaluates the format string, it accesses the extra parameters given to the function. However, there is a special format parameter in *ANSI C* called '*%n*', which can write the number of bytes printed so far to the specific memory. Because parameters and other important program data are all stored on the stack, if the format string can be controlled by attackers, they can overwrite returned addresses or other data they want [30].

IV. Command Injection

Like format string, if an uncontrolled input string is directly passed to an OS execution system call or a shell execution function, attackers can easily execute system commands by injecting malicious strings into the input.

E. DLL Injection

DLL injection is a technique used for running code within the address space of another process by forcing it to load a dynamic-link library (DLL) [37]. The injected code can hook the system or library calls, such as *system* or *malloc*, without modifying any existing programs. We could interrupt programs by the injected code and analyze the symbolic relationship between arguments and inputs. Besides standard libraries, we can also hook third-party libraries. The functions we hook in this paper are shown in **Table 2**.

On Linux (or other Unix-like OS), arbitrary libraries can be linked to one's custom library by setting the *LD_PRELOAD* environment variable[1]. Such a library can be created with *GCC* by compiling with -*fPIC* option[2] and linked with -*shared* option[3]. On Windows, there are multiple ways to do this, one of which is the hooking call *SetWindowsHookEx*[4]. Our work focuses on Linux platform, but is easy to extend.

[1] http://www.kernel.org/doc/man-pages/online/pages/man8/ld-linux.so.8.html

[2] http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Code-Gen-Options.html#Code-Gen-Options

[3] http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Link-Options.html#Link-Options

[4] http://msdn.microsoft.com/en-us/library/ms644990.aspx

CHAPTER 3 Method

Unlike traditional fuzzers generating crash inputs (shown in **Figure 5**), our work *CraxFuzzer* generates proof-of-concept (POC) exploit instead. A program crashes because its instruction pointer EIP is abnormal. Hackers want to find crashes because if EIP is modified during program execution, they may have chances to control the program by changing EIP to the shell code address. If we have some ways to overwrite the returned address, i.e., EIP on the stack to control the program, a crash situation is not necessary. For example, in an uncontrolled format string case, the ability of generating a crash is still far easier than generating an exploit. In a command injection or a SQL injection problem, crashing a program is not even of our primary consideration. That is the concept of POC generation, reducing the step of finding crashes and focusing on watching sensitive functions that may directly or indirectly affect program execution.

There are four stages in our fuzzing framework; *test case acquisition*, *target searching*, *proof-of-concept generation*, and *verification*. The architecture overview is shown in **Figure 6**. In test case acquisition, we acquire test cases from traditional fuzzers, the Internet, specs of the program, or a path exploration which will be merged to symbolic execution in the next stage. In target searching stage, we use a null-constraint single-path concolic execution method to search for our target functions. In POC generation, we generate inputs that would crash the program or affect the sensitive arguments, and then run it again to verify in the final step.

To make our proposed methods clear and easy to understand, we use an open-source software *XMail* [21] to explain them in **Section 3-E**, and a detailed comparison of traditional fuzzers with our work is provided. At the end of this chapter, we will show the implementation details of how we build our target-aware symbolic execution framework on *S2E*, including modules and implemented sensitive functions.

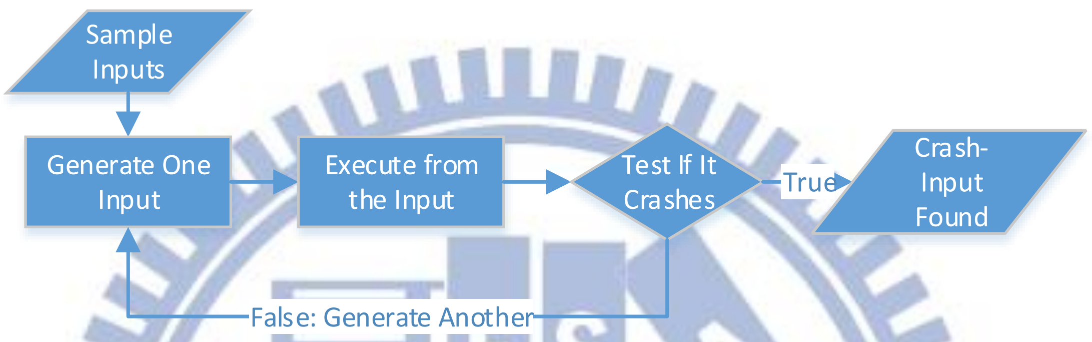


Figure 5: Traditional Fuzzer Architecture Overview

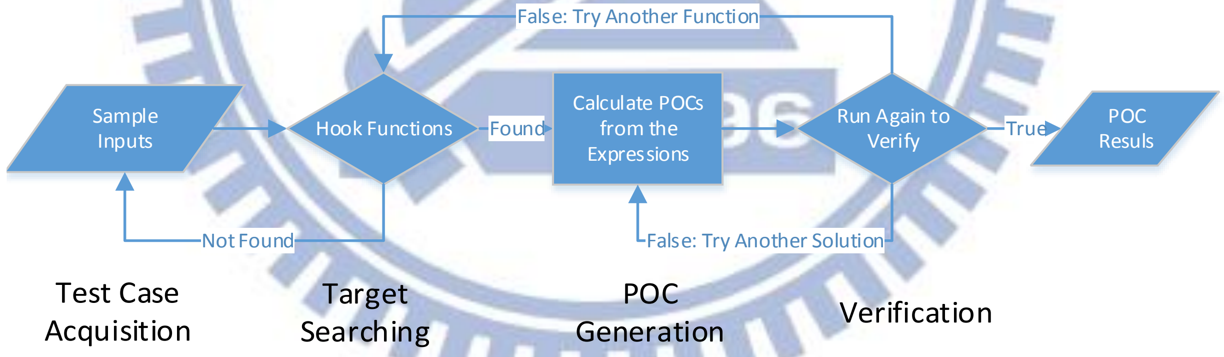


Figure 6: Target-Aware Symbolic Execution Framework Overview

Now we are going to explain our methods step by step in detail as follows.

A. Test Case Acquisition

To test a program, a sample input is needed as a seed to induct the mutation. Otherwise, it will cost too much to generate inputs from nothing. There are multiple ways to acquire test cases. Downloading inputs from the Internet or manufacturing a new one from the specs are straight-forward methods. If these samples cannot satisfy the requirements, we can generate more of them by traditional fuzzing or symbolic execution. According to our experiences, a regular input is usually good enough to find serious bugs. We will demonstrate it in **Chapter 4**.

B. Target Searching

We use symbolic execution to explore paths and search for our target functions. A schematic diagram of this execution is shown in **Figure 7**. The method of null-constraint single-path concolic execution has been introduced in **Section 2-B**. We want to check whether there is any sensitive function whose arguments are also symbolic or not by hooking the standard libraries. The techniques of how to hook a function call has been discussed in **Section 2-E.** We check every function it hooked and its arguments while executing the program. The hooked functions, or the target functions, we chose are usually dangerous and may cause vulnerabilities. If an argument, e.g. an input of a function, has been detected symbolic, luckily we have found the fragile part of this program. The input bytes corresponding to the symbolic argument are called ***hot bytes***, which means these bytes can directly influence this sensitive function and have a good chance to exploit. More discussions of hot bytes will be made in **Section 4-B**. We can also print call stacks of the program to help developers debugging.

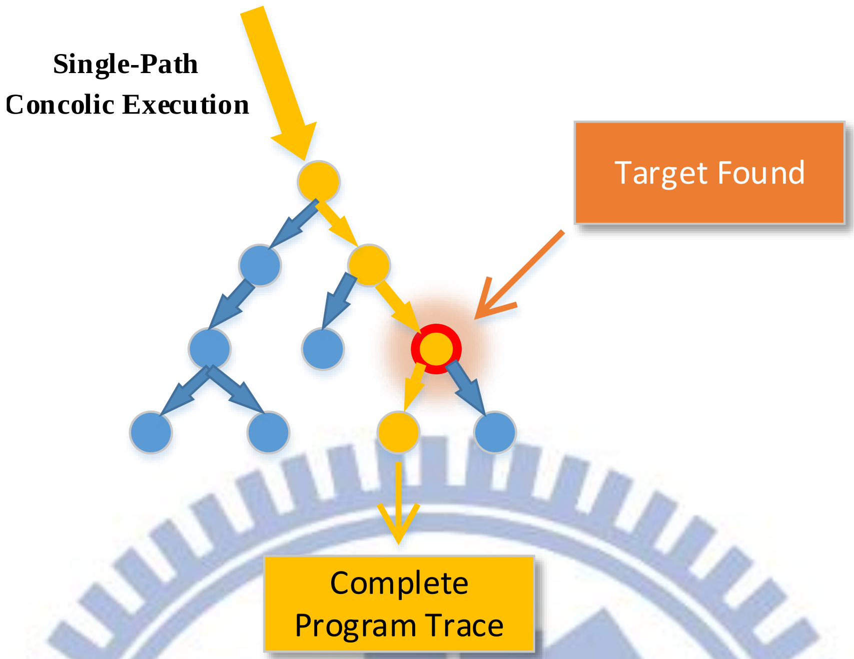


Figure 7: Target Searching Overview

```
x = read()
x = x + 1
malloc( 2 * x )
```

Figure 8: Transformation Function Example

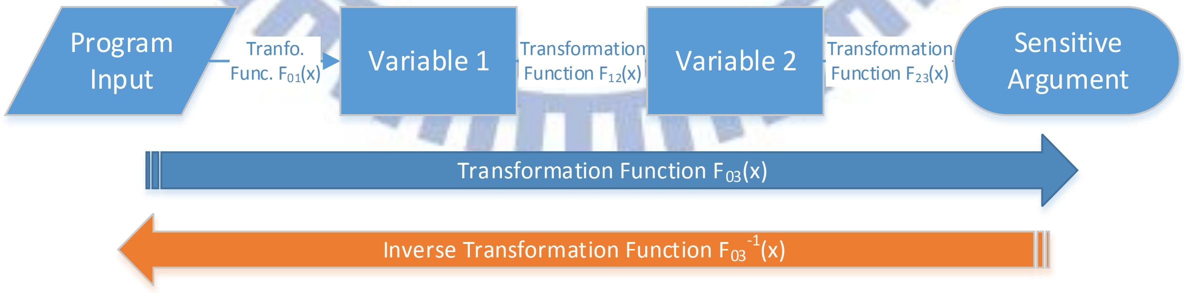


Figure 9: Transformation Relations in Program Execution

C. Proof-of-Concept Generation

The sensitive situation is a formula in terms of symbolic information. In a large program, a value often undergoes a series of transformation between being read as input and triggering a potential vulnerability. For example in **Figure 8,** the argument *malloc* in **line 3** has been passed to two transformation functions, $f(x) = x + 1$ and $f(x) = 2 * x$. Once a sensitive situation has been confirmed, we can generate the proof-of-concept (POC) exploit, which is an evidence that we already have the ability to exploit this program. To find the input that makes the variable to be a certain value, we need to solve the inverse of the transformation functions. In **Figure 9**, we can set *Transformation Function* $F_{03}(x)$ to be a constant value C, which we want the sensitive argument to be. Use a constraint-solver such as *STP* [13] to solve equation $F_{03}(x) = C$, e.g. find the results of $F_{03}^{-1}(C)$. This is one of functionalities of symbolic execution. Note that in the null-constraint symbolic execution, we do not record any path constraints, which means the results may not be feasible (but efficient). We will discuss about this in **Section 3-D** and in **Chapter 4**.

According to the properties of each sensitive situation, we can classify the problems into designated types as follows.

I. Formats in Format Functions

If a format argument in format functions, *prinf* and *syslog* for example, is symbolic, which means it can be affected by input bytes, there is a high possibility that a format string bug may exist in the testing program. If the developers have not checked the conversion specification character "%" or restricted the length of the inputs, attackers may exploit this program by a well-designed input that contains shell codes and the conversion specifier "%n" [30].

II. Commands in Execution Functions

Command arguments in execution functions such as *system* and *exec* are sensitive too. If a command segment passes to these functions without being checked, attackers could use "*&&*", "*;*", or "*&*" as conjunctions of normal and malicious commands, or directly inject a malicious binary to hack this system.

III. Sources in Memory Copy Functions

If the source argument in memory copy function can be affected by input bytes, there is a chance for hackers to generate buffer overflows in our testing program by increasing the length of our tainted input bytes. Although some memory functions use a size argument to restrict size of buffers to be copied, developers often forget to check the destination buffer size or the end of string character "*\0*". Lots of buffer overflow vulnerabilities have been revealed by *CVE* mentioned as above, and we will demonstrate some of them in **Chapter 4**. *Str(n)cpy*, *read*, and *memset* are three functions which belong to this type.

IV. Lengths in Memory Copy Functions

Similar to sources in memory functions, controlling the length of the function is more intuitive. Without proper checking, a buffer overflow is easy to happen if the length argument and buffer size is inconsistent.

V. Sizes in Memory Allocation functions

If a size argument in memory allocation functions is symbolic, there may be an integer overflow in our testing program. If we make the size (an integer) overflow, the operating system will allocate less memory space than the program wants, and it may use the memory out of the boundary in the heap, which causes a heap-based buffer overflow, e.g., *malloc(0xFFFFFFFF)*.

D. Verification

After generating a new input without path constraints, we need to check if the input is feasible. We do this by re-running the program with the new input and checking whether the sensitive argument is rewritten or not. If the answer is yes, we have almost confirmed that we can change the sensitive argument to an arbitrary value including an exploit. We call the generated input a proof-of-concept exploit because we have proved that there is a vulnerability and explained how to manipulate it in this situation. If the answer is no, which means the input may not be feasible due to the changing of the program's running path, we need to add some path constraints or change the previously used value to calculate a new input. Once no input can be generated, we have to go back to the first step acquiring a new sample input.

E. Example: *XMail-1.21*

XMail [21] is a lightweight email server in comparison with traditional mail servers. In 2005, a stack-based buffer overflow vulnerability in module *sendmail* in *XMail* has been revealed and numbered *CVE-2005-2943*. In this bug, remote attackers can execute arbitrary code via crafting the specific field in an email letter.

```
From: hchung@cs.nctu.edu.tw
To: hchung@cs.nctu.edu.tw
Subject: A Target-Aware Symbolic Execution Framework for Fuzz Testing

This is an XMail test letter

```

Figure 10: Sample Email with Envelopes for *XMail*

```
In SendMail.cpp:

341 static char const *AddressFromAtPtr
        (char const *pszAt, char const *pszBase, char *pszAddress)
342 {
344     char const *pszStart = pszAt;
351     char const *pszEnd = pszAt + 1;
…
355     int iAddrLength = (int) (pszEnd - pszStart);
357     strncpy(pszAddress, pszStart, iAddrLength);
358     pszAddress[iAddrLength] = '\0';
359
360     return (pszEnd);
362 }
```

Figure 11: Function *AddressFromAtPtr* in *SendMail.cpp* in *XMail*

Figure 10 shows a sample email we commonly used in our daily life. This original email contains four fields, including sender, receiver, subject, and body, which the mail transfer agent (MTA) actually delivers. We make this email symbolic and pipe it into our testing program. In target searching stage, we found a sensitive function *strncpy* has been hooked and its source-string argument is symbolic. From tracking the symbolic information, this argument is directly affected by bytes from 0x20 to 0x34, and the corresponding string is "*hchung@cs.nctu.edu.tw*", the receiver's email address which we marked gray in **Figure 10** and called *hot bytes*. We look into the source code in **Figure 11** to confirm our guest. In **line 357**, *strncpy* copies a string which is read from the letter to a fixed size array (read the complete source code and you will know). Now we know the receiver's address will be copied to a certain buffer. That is a buffer overflow suspect. We use the expressions (shown in **Figure 12**, which merely extend byte to word) gathered before and our buffer overflow heuristics to decide a long receiver's address field. Then fill it back to the input and run it again. A segmentation fault exception will be raised and that verifies our result.

```
(Concat w32 (Extract w8 24 N0:(SExt w32 N1:(Read w8 0x20 INPUT)))
            (Concat w24 (Extract w8 16 N0) (Concat w16 (Extract w8 8 N0) N1)))
(Concat w32 (Extract w8 24 N0:(SExt w32 N1:(Read w8 0x21 INPUT)))
            (Concat w24 (Extract w8 16 N0) (Concat w16 (Extract w8 8 N0) N1)))
(Concat w32 (Extract w8 24 N0:(SExt w32 N1:(Read w8 0x22 INPUT)))
            (Concat w24 (Extract w8 16 N0) (Concat w16 (Extract w8 8 N0) N1)))
…
```

Figure 12: *Strncpy* Argument Expressions of *INPUT* in *XMail*

However, if we turn over this work to traditional fuzzers, they have to mutate the input mail from byte to byte, which has a large time complexity of ***O****(2^n)*. Even if we know the format of the letter, which is hard to be formulized, we cannot guarantee that the mutated inputs could cover the problematic code. The comparison of traditional fuzzers with our work is listed in **Table 1**.

Table 1: Comparison of Traditional Fuzzers with *CraxFuzzer*

	Traditional Fuzzers	***Traditional Symbolic Fuzzers***	***Our Target-Aware Symbolic Fuzzer***
Targets	Exceptions (crashes)	Exceptions (crashes)	Sensitive functions
How to Find	Generate inputs	Explore paths	Hook functions
How to Verify	Debugging tools	Debugging tools	Check hooked funcs.
Seed Inputs	Existing inputs	X	Existing inputs
How to Generate New Inputs	Mutate seed inputs	Solve path constraints	Only one input
Hit Rate of an Input to Reach a Target	Very low	Low	High (hooked funcs. are easier to find)
Result Types	Crash inputs	Crash inputs	POC Exploits
How to Generate Results	Mutate seed inputs	Solve path constraints	Solve constraints from expressions
Results Generation time	Fast	Slow	Only one input
When to Generate Results	Before execution	After targets being found	After targets being found
Results Accuracy	Very high	Not too high (confli-ctive constraints)	Not too low (ignored path constraints)
Aware of Constraints and Expressions	X	O	O

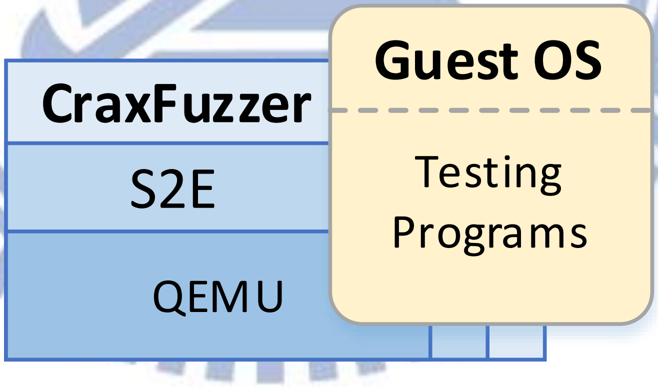


Figure 13: *CraxFuzzer* Architecture

F. Implementation

Our fuzz testing framework, *CraxFuzzer* (shown in **Figure 13**), is built on top of *S2E*, which is discussed in **Section 2-C**. Identical to *S2E*, *CraxFuzzer* applies symbolic execution via a virtualizer, e.g. *QEMU*. We can test programs on different systems by this virtualization technique. Testing programs, generated sample inputs and system configurations will be sent to the guest OS after *CraxFuzzer* starts. The biggest defect of this framework is the lack of support for floating points due to the implementation of *KLEE*.

Table 2: List of Discovered Sensitive Functions

Sensitive Functions	*Sensitive Arguments*	*Vulnerable Situations*
fread	Length	Integer/Buffer Overflow
read	Length	Integer/Buffer Overflow
memset	Length	Integer/Buffer Overflow
memcpy	Source, length	Integer/Buffer Overflow
strcpy	Source	Buffer Overflow
strncpy	Source, length	Integer/Buffer Overflow
syslog	Format	Format String
vfprintf	Format	Format String
vsnprintf	Format, length	Format String, Integer/Buffer Overflow
sprintf	Format	Format String Buffer Overflow
fprintf	Format	Format String
system	Command	Command Injection
exec family	Path (file)	Command Injection
realloc	Size	Integer/Buffer Overflow
malloc	Size	Integer/Buffer Overflow

In guest OS, we implement a wrapper to inject our custom libraries and redirect the program inputs of every testing object. The custom libraries are used to overwrite sensitive functions in standard libraries. According to our experiences and some researches [10, 30, 42, 44], we intercepted sensitive functions listed in **Table 2.** In a hooked function call, we use custom CPU op code to check whether an argument is symbolic, printing the expressions and call stacks. The input redirection is used to symbolize inputs. For programs whose input is read from standard input (*stdin*) or arguments, it is easy to symbolize parts of them or all of them. For programs whose input is read from files, we have to map files in the hard drive to the memory first by system call *mmap*. However, files are often large in size. We choose adaptive file segments described in **Section 2-B-I** to symbolize important data structures only.

In host OS, log parser, input generator and verifier are implemented. In the hooked function call, we output symbolic expressions of the sensitive arguments. The parser is used to analyze this output. It determines whether this sample input is useful, generating new inputs based on constraints and heuristics described in **Section 3-C**. After a possible exploit POC is generated, we re-run it symbolically to verify due to the deduction of constraints discussed in **Section 2-B-III**.

We build the null-constraints-single-path-concolic-execution technique into *S2E* plugins. These plugins contain the constraints reduction and loggers. The coordinator of these components is also implemented in a plugin. Relations between components are shown in **Figure 14**. There are two cycles in this figure. The first one is about finding sensitive arguments and its corresponding sensitive input bytes. The second one is about generating POC. If any of these steps fails, this framework is able to fall back and try again until there are no available sample inputs anymore.

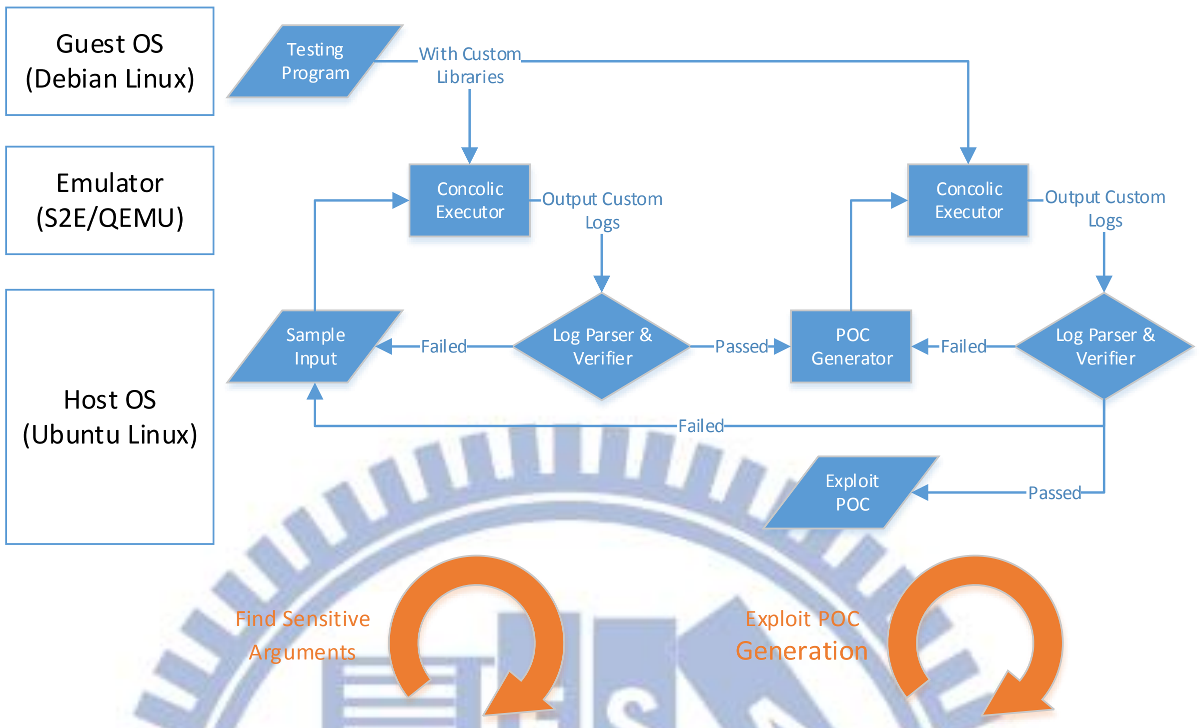


Figure 14: *CraxFuzzer* Component Relations

CHAPTER 4 EVALUATION

In this chapter, we present four sets of experiments and our experiment environment. In **Section 4-B**, we evaluate the value of hot bytes concept to reduce testing space by observing the ratio of hot bytes to total input bytes. In **Section 4-C**, we present the adaptive-input technique, which utilizes hot bytes, headers, or other information to make part of input symbolic. In software testing, if we test only the most important parts of the input, redundant testing could be omitted. In **Section 4-D**, we introduce the results of null-constraint technique, which could speed up symbolic execution in some specific situations. And in **Section 4-E**, we show some vulnerabilities we detected in popular applications. At the end of this chapter, we give some real-world case studies, including *sudo* and *tiff*, to demonstrate the power of our work.

Table 3: List of Testing Applications

Application	*Version*	*Operation*	*Input Type*	*Line of Code*
XMail	1.21	Send email	Stdin	31480
Sudo	1.8.0	./sudo -D9	Arguments	25324
Exim	1.21	./exim -bh ::%eth0	Arguments	64102
Socat	1.4	./socat –lyAAAA	Arguments	13929
Ncompress	4.2.4	./ncompress FILE	Arguments	1432
Gif2png	2.5.3	./gif2png FILE	Arguments	1353
Iwconfig	V26	./iwconfig eth0	Arguments	5285
Tipxd	1.1.1	./tipxd –fFILE	Arguments	1066
Mcrypt	2.6.8	Decryption	Files	5846
Tiff	3.6.1	View TIFF image	Files	42077
Tiff	3.7.0	View TIFF image	Files	43717
Mplayer	1.0rc2	Play VQF media	Files	488618
Vim	1.7	Open file with filetype.vim	Files	377767

A. Experiment Setup

Although *CraxFuzzer* could test programs on different OSs, we are dedicated to study software on Linux platform. We apply *CraxFuzzer* to a large number of real-world applications. A list of these applications is summarized in **Table 3.** All of them are popular open-source projects for Linux. The "Operation" column shows what operations or what running arguments we take to test these applications. Note that the inputs are common and easy to be retrieved from the Internet or the manual pages. If further researchers want to focus on testing some specific libraries such as OpenFlow Applications [6], PHP SQL libraries [48], or even the CPU register EIP [18], they could just extend this work by adding these libraries to our sensitive function pool and defining the vulnerable situations they want the program to be.

Our experiments are conducted on a virtual machine with Intel Xeon E3-1230V2 at 3.3 GHz and enough memory, running Ubuntu 12.04 64 bit. The guest OS is running Debian 6.0.6 32 bit.

Table 4: Hot Bytes Identification Results

Application	*# Symbolic Bytes*	*# Detected Hot Bytes*	*Run Time*
XMail	156	22	5s
	368	40	6s
Tiff-3.6.1	18278	5	6s
	46271	3	6s
Tiff-3.7.0	18278	158	8s
	46271	130	10s
Vim	1437	258	113s
Mplayer	1024/120132	1024	100s

B. Hot Bytes Identification

In this experiment, we measure the ratio of hot bytes to total input bytes and the complete execution time. We feed common inputs, which are retrieved from the Internet or the spec, to some input-file-based testing programs in. The hot bytes are some segments of the input that directly affect the sensitive arguments listed in **Table 2**.

Table 4 shows the results of four common open-source Linux applications. *XMail* is an email tool, whose input is a plaintext letter. *Tiff* is a library that manipulates *TIFF* files. *Vim* is a common editor, tested with one of its scripts called *filetype.vim*. *Mplayer* is a media player that plays various kinds of media. The "Operation" column represents what we do to test this application, the "# Symbolic Bytes" column represents how many bytes of the total input we make symbolic and "# Hot Bytes" indicates how many bytes of the input are sensitive. The total run time is shown in the last field "Run Time".

Figure 15, **Figure 16** and **Figure 17,** shows hot bytes distributions for *tiff-3.7.0* tests and *vim* test respectively. The first *TIFF* file can be downloaded from here[1] and the second file can be found here[2]. Character 'H" denotes a hot byte and "dot" denotes a non-hot-byte. As the results show, all hot bytes are centralized at the beginning of the file or at the end of the file, which means they are probably headers.

The ratio of hot bytes to total inputs is approximately in the range from 1% to 20%. That means if we want to find vulnerabilities in **Table 2**, we have to test

[1] http://www.fileformat.info/format/tiff/sample/3794038f08df403bb446a97f897c578d/download

[2] http://www.fileformat.info/format/tiff/sample/e9898d47632440288d86553edf676007/download

only 1% bytes over the whole input in the best case, which reduces the testing input space from 8^N to $8^{N/100}$. In the *mplayer* case, almost all bytes are hot bytes and are hooked in *memset* functions. We think the reason is that while playing media, the player will copy the input to buffers. We believe that if we reduce the number of kinds of sensitive functions, *memset* in the *mplayer* case for example, we can obtain a more accurate hot bytes distribution. In some cases, we do not make all the input symbolic because we want to avoid bytes from involving floating operations or the input space is too large. We use “Symbolic Bytes/Total Bytes” to denote this situation. As we count the hot bytes in an acceptable running time, this technique can reduce a lot of execution time.

The related work to be compared with would be *TaintScope*. However, the *ImageMagick* test picked from *TaintScope* failed because of the *KLEE* floating point issue. And the rest results of *TaintScope* cannot be taken because they are based on Windows platform, which is not supported.

```
Skipped 17700 dots
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
.......H HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH
HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH
HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHH... ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ......
```

Figure 15: Hot Bytes of TIFF File I

```
HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH
HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH
H....... ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
Skipped 45700 dots
```

Figure 16: Hot Bytes of TIFF File II

```
HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH
HHHHHHHH .HHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH
HHHHHHHH HH..HHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH HHHHHHHH
HHHHHHHH HHHHHHHH HHHH.... ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
......H. ........ ........ ..HHHHHH HHHHHHHH HHHHHHHH HHHHHHHH H.......
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ....HH.. ......H. .H....H. H...H... ........ ..H..H..
.H...H.. H..H..H. .H...H.. ..H..... ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ ........ ........ ........ ........ ........
........ ........ ........ .....
```

Figure 17: Hot Bytes of File Using *Filetype.vim*

C. Adaptive-Input Technique Evaluation

With header information retrieved from hot bytes technique and other documents of file format, we are able to locate the important part of input. For files whose hot bytes can be easily identified, we can symbolize only the hot bytes parts because hot bytes are the only inputs that affect sensitive functions. For files in which hot bytes cannot work, we have to read documents to find the important part, which is usually the header, additionally.

Results of this experiment are shown in **Table 5**. The "# Input Bytes" means the number of total input bytes while "# Adaptive Symbolic Bytes" means the number of the important bytes we believe to be. The "Regular Run Time" column represents the time cost of running symbolic execution with all bytes symbolic while the "Adaptive-Input Run Time" column represents running time using this technique. We can see that the running time is improved if the same hot bytes are found.

For cases which take a long time like *mplayer*, we may use the concept of divide-and-conquer to run multiple tiny segments (1024 bytes in this case) respectively and then combine the results.

Table 5: Adaptive-Input Technique Results

Application	*# Input Bytes*	*# Adaptive Symb. Bytes*	*Run Time (Regular)*	*Run Time (Adaptive-Input)*
XMail	156	22	5s	4s
	368	40	6s	5s
Tiff-3.7.0	18382	180	8s	5s
Vim	1437	583	113s	15s
Mplayer	120132	1024	X	100s

D. Null-Constraint Technique Evaluation

In this experiment, we compare the execution time with or without gathering path constraints. Results are shown in **Table 6**. The "Regular Run Time" column represents the time cost of running symbolic execution with path constraints while the "Null-Constraint Run Time" column represents running time using this technique. The "# Symbolic Bytes" column shows number of bytes we symbolize. The "# Symbolic Bytes" column in the "sudo" row is uncertain because it is determined by user's system. The number of bytes is approximated in a level of ten to the first power.

After a possible malicious input is generated by our null-constraint technique, we need to verify this input again due to the lack of path constraints. We can choose any set of constraints and put them back to constraint-solver for re-running. In some cases like hot bytes identification, the path constraints are even useless at all and does not have to be recorded. And of course it reduced the testing input space.

We notice that there exists a significant difference between executing with or without constraints in running time in some applications. For large software with complicated calculation such as *mplayer* and *vim*, we can have a great improvement on it. However, for simple applications whose running time is short, the efficiency enhancement is limited. These experiments affirm our hypothesis.

Table 6: Null-Constraint Technique Results

Application	*# Symbolic Bytes*	*Run Time (Regular)*	*Run Time (Null-Constraint)*
XMail	156	6s	5s
	368	10s	6s
Tiff	18278	7s	6s
Sudo	All arguments	10s	8s
Vim	1437	775s	113s
Mplayer	1024/120132	Over 500,000s	100s

Table 7: CraxFuzzer Fuzzing Results

Application	*Advisory ID*	*Vulnerability Type*	*Hooked Func.*	*# Symbolic Bytes*	*Run Time*
XMail	CVE-2005-2943	Buffer Overflow	Strncpy	156	5s
Sudo	CVE-2013-1775	Format String	Vfprintf	All args.	8s
Exim	EDB-ID#796	Buffer Overflow	Strncpy	All args.	5s
Socat	CVE-2004-1484	Format String	Syslog	All args.	5s
Ncompress	CVE-2001-1413	Buffer Overflow	Strcpy	All args.	5s
Gif2png	CVE-2010-4695	Buffer Overflow	Strcpy	All args.	5s
Iwconfig	CVE-2003-0947	Buffer Overflow	Strcpy	All args.	5s
Tipxd	OSVDB-ID#12346	Format String	Syslog	All args.	5s
Mcrypt	CVE-2012-4409	Buffer Overflow	Fread	All args.	6s
Tiff	CVE-2004-1307	Integer Overflow	Malloc	18278	5s
Mplayer	CVE-2008-5616	Buffer Overflow	Memcpy	1024	100s
Vim	CVE-2008-2712	Command Injection	Execvp	1437	113s

E. Fuzzing Results

Table 7 lists well-known vulnerabilities we have found yet and the execution time is also provided. The "Advisory ID" column shows advisory identifier information. CVE-YYYY-0000 presents CVE identifiers while the eight digits are year and series number of this vulnerability, EDB-ID presents Exploit-DB[1] identifiers and OSVDB-ID presents Open Sourced Vulnerability Database[2] identifiers. The "Vulnerability Type" shows the corresponding type introduced in **Section 2-D**. The "Hooked Function" columns shows which function call we have hooked and found the corresponding vulnerability successfully in the standard library. The "Run Tine" column shows the execution time to generate the exploit

[1] http://www.exploit-db.com/

[2] http://osvdb.org/

POC of this vulnerability.

First we discuss about the running time. Applications with less computation or the less symbolic input have a short execution time. Five seconds is almost the set up time of our framework with the time cost of booting images to the specific snapshots. The running time of symbolic execution is short and can be ignored. For large applications such as *tiff*, *vim* and *mplayer*, the execution time is also acceptable. Compared with traditional fuzzers, which generate millions of test cases and execute millions times, they may take days to find only crashes for these applications and are not promised to be exploitable.

Many bugs we have found are very simple. They are often caused by the lack of checking buffer boundaries of memory copy functions or escaped characters of format string functions. Although some developers use *strNcpy* series functions to restrict bytes to be copied, they forget to check source bytes and destination bytes simultaneously. For vulnerabilities whose input is composed of readable strings, the transformation during program execution is usually linear transformation, which is simple to exploit. The uncontrolled format string, command injection and SQL injection cases belong to this type. For software with complex structures and systematic quality testing, our proposed method can easily outperform existing methods. Case studies in next section will introduce some examples, one of which includes complicated transformations.

F. Case Studies

We introduce two real-world cases to demonstrate *CraxFuzzer* in this section. Both of them are well-known applications and contain serious security issues.

```
In sudo.c:

1097 void
1098 sudo_debug(int level, const char *fmt, ...)
1099 {
1100     va_list ap;
1101     char *fmt2;
1102
1103     if (level > debug_level)
1104    return;
1105
1107     easprintf(&fmt2, "%s: %s\n", getprogname(), fmt);
1108     va_start(ap, fmt);
1109     vfprintf(stderr, fmt2, ap);
1110     va_end(ap);
1111     efree(fmt2);
1112 }
```

Figure 18: Parts of *Sudo* Code

I. *Sudo-1.8.0*

Sudo is a commonly used system utility that can execute a command as another user, especially administrator. Therefore, security issues of *sudo* are deeply concerned about. In **line 1107** and **line 1109** in **Figure 18**, there is a format string function which takes its program name as format argument, which is an uncontrolled format string problem. As program name is one of the program execution arguments, which is also program inputs, symbolic information can be intercepted in *vfprintf* with proper sample inputs. These sample inputs can be derived from the manual page, *-D* flag for example. This vulnerability is announced in CVE website with a CVE-2013-1775 identifier.

```
In tif_dirread.c: (Simpilified)

cp = (char*)malloc(nstrips * sizeof (uint32))
```

Figure 19: Parts of *Tiff* code

```
SymbExpression malloc_size -
    (Shl w32 (Extract w32 0 (UDiv w64 (ZExt w64 (Add w32 (w32 0xffffffff)
    (Add w32 N0:(ReadLSB w32 0x72 v0_file_0)(ReadLSB w32 0x2a v0_file_0))))
    (ZExt w64 N0)))(w32 0x2))
SymbExpression malloc_size - Value: 0xc
```

Figure 20: Report of *Malloc* of *Tiff* Execution

II. *Ttiff-3.6.1*

Integer overflow in the *TIFFFetchStripThing* function in *tif_dirread.c* for *libtiff 3.6.1* allows remote attackers to execute arbitrary code via a TIFF file with the *STRIPOFFSETS* flag and a large number of strips, which causes a zero byte or a small bytes buffer to be allocated and leads to a heap-based buffer overflow. This vulnerability is registered as CVE-2004-1307.

The sensitive function *malloc* we have found is shown in **Figure 19**. Variable *nstrips* denotes the number of strips of the TIFF file. We want to make expression "*nstrips * sizeof(uint32)*" to overflow as zero or a small value. There are many transformations from program start to the target function. However, the transformation functions of *nstrips* variable are too hard to be found out by eyes. We can use *CraxFuzzer* to do this. The report is shown in **Figure 20**. This report shows the results of executing TIFF library by feeding the 18 KB file we have downloaded from the Internet and introduced before. The results can be simplified to this equation

*SIZE = (A+B-1)/B*4*

where A is INPUT[0x2A] and B is INPUT[0x72].

That means the 0x2Ath byte and 0x72th byte of our input can make this *malloc*'s size argument be the value of 0xC. If we set this size argument to be zero or a small value and solve a set of integer overflow answer of *A* and *B*, a heap overflow problem may occur. We get the answer that *A* equals to 0x40000000 and *B* equals to 0x1 by solving equation "*(A+B-1)/B*4=0*". We then overwrite them to the original input and run it again to verify. It crashes and the value of size argument has been changed. This proves that we can find a way to control the heap overflow of *tiff-3.6.1* to cause the program buffer overflow by feeding a normal input downloaded from the Internet.

CHAPTER 5 CONCLUSION

In this thesis, we present *CraxFuzzer*, a target-aware directed whole-system symbolic fuzzing framework. By using libraries hooking and hot bytes identification techniques, *CraxFuzzer* can locate sensitive parts of the program after being fed a regular input, and generate the corresponding POC exploit efficiently. *CraxFuzzer* can dramatically reduce the testing space compared with traditional fuzzers and find conditions they are not able to reach, helping developers to find the security vulnerabilities and to fix them in a short time. For cases whose source codes are available, it is also possible to use debug information to print out the call stack and other information. We have applied *CraxFuzzer* to 17 previously known issues of different security types. Experimental results show that it can accurately locate the sensitive parts and greatly improve the effectiveness of fuzz testing.

TaintScope [42], which inspires us, provided the concept of finding hot bytes by dynamic taint analysis. However, our methods are easier and more straightforward by taking advantages of the property of symbolic execution. *Splat* [46] defines a buffer overflow situation and *Catchconv* [29] defines an integer conversion error situation, while *IntScope* [44] defines an integer overflow situation and *Saxena, P., et al.* [33] defines a loop-extended situation. At the application level, *NICE* [6] models OpenFlow applications to find network bugs. In spite of that, all of them merely focus on one specific condition, and it is hard to generalize the targets. *Caselden, D., et al.* [7], *McCamant, S., et al.* [25] and *STrigger* [22] introduced vulnerability-condition-based, or trigger-condition-based test case generation methods respectively. Nevertheless, because of the different symbolic engines and heuristics they use, source code is needed and the testing tools are not platform-independent and whole-system.

If researchers want to modify our framework to create real exploits, it is easy to implement by replacing POC with shell code. Shell code is machine or OS dependent and more complicated to be generated automatically, and that's why our work delegate this step to other tools. *CRAX* [18, 47, 48], *AEG* [3] and other researches [6, 7, 16] are dedicated to this field.

However, there are several limitations in the current implementation of *CraxFuzzer*. First, the lack of floating support of *KLEE* is a big problem to test programs with floating point operations. It will terminate *S2E* if countering this situation. We can use adaptive-input technique to strategically avoid this implementation problem. Second, due to the complex path constraints and the natural property of hash functions, checksum, cryptographic operation or digital signature, which are designed to protect against data alteration, are not recommended being tested by our work. *TaintScope* [42] has a great success on checksum reconstruction. It is possible to combine these modules. Third, our searching mechanism is based on hooking functions. If we want to find a vulnerability that is not related to functions (**Figure 21**, for example), the DLL injection framework does not work anymore. Some new mechanisms must be found to generalize this situation. However, with vulnerabilities that are related to functions, such as PHP SQL libraries [48], *CraxFuzzer* works pretty well. It is also possible to hook CPU register EIP [18] to find the condition of crash. Forth, path constraints selection is also a challenge. What path constraints to pick is a Satisfiability (SAT) problem, which is very hard [41]. It would take great efforts to do it well. The last weak point is how to retrieve a sample input that can lead us to the bug. In most cases, regular inputs cannot be executed into the problematic code due to the imperfect of unit test cases. We must generate such inputs ourselves. Control flow graph (CFG) and call graph are good mediums for use [44]. We can use shortest-path algorithms to find a path to the sensitive function and generate the corresponding input like *STrigger* [22]. There are also lots of test case generation researches to help us creating inputs [32].

```
while(n--)
  str[n] = 0;
```

Figure 21: Vulnerability Not Related to Functions Example

References

[1] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, ed: Springer, 2008, pp. 367-381.

[2] W. A. Arbaugh, W. L. Fithen, and J. McHugh, "Windows of vulnerability: A case study analysis," *Computer,* vol. 33, pp. 52-59, 2000.

[3] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic Exploit Generation," in *NDSS*, 2011, pp. 59-66.

[4] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41-46.

[5] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *OSDI*, 2008, pp. 209-224.

[6] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A NICE way to test OpenFlow applications," *NSDI, Apr,* 2012.

[7] D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant, and D. Song, "Transformation-aware exploit generation using a HI-CFG," *University of California, Berkeley, Tech. Rep. UCB/EECS-2013-85,* 2013.

[8] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *ACM SIGARCH Computer Architecture News,* vol. 39, pp. 265-278, 2011.

[9] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS),* vol. 30, p. 2, 2012.

[10] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, 2000, pp. 119-129.

[11] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ed: Springer, 2008, pp. 337-340.

[12] M. Eddington. (2011). *Peach fuzzing platform*. Available: http://peachfuzzer.com/

[13] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification*, 2007, pp. 519-531.

[14] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing,"

in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, 2009, pp. 474-484.

[15] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated Whitebox Fuzz Testing," in *NDSS*, 2008, pp. 151-166.

[16] S. Heelan, "Automatic generation of control flow hijacking exploits for software vulnerabilities," University of Oxford, MSc Computer Science Dissertation, 2009.

[17] S. Hocevar. (2011). *zzuf—multi-purpose fuzzer*. Available: http://caca.zoy.org/wiki/zzuf

[18] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong, "CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, 2012, pp. 78-87.

[19] J. C. King, "Symbolic execution and program testing," *Communications of the ACM,* vol. 19, pp. 385-394, 1976.

[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, 2004, pp. 75-86.

[21] D. Libenzi. *XMail*. Available: http://www.xmailserver.org/

[22] J. Liu, Q. Wei, Q.-x. Wang, and T. Guo, "Trigger condition based test generation for finding security bugs," in *Systems and Informatics (ICSAI), 2012 International Conference on*, 2012, pp. 1106-1110.

[23] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Static Analysis*, ed: Springer, 2011, pp. 95-111.

[24] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey, "2011 CWE/SANS Top 25 Most Dangerous Software Errors," *Common Weakness Enumeration,* vol. 7515, 2011.

[25] S. McCamant, M. Payer, D. Caselden, A. Bazhanyuk, and D. Song, "Transformationaware symbolic execution for system test generation," Tech. Rep. UCB/EECS-2013-125, University of California, Berkeley (Jun 2013)2013.

[26] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM,* vol. 33, pp. 32-44, 1990.

[27] C. Miller, J. Caballero, N. M. Johnson, M. G. Kang, S. McCamant, P. Poosankam*, et al.*, "Crash analysis with BitBlaze," *at BlackHat USA,* 2010.

[28] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *Proceedings of the 18th*

conference on USENIX security symposium, 2009, pp. 67-82.

[29] D. A. Molnar and D. Wagner, "Catchconv: Symbolic execution and run-time type inference for integer conversion errors," *UC Berkeley EECS,* 2007.

[30] T. Newsham, "Format string attacks," ed, 2000.

[31] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[32] J. Röβler, G. Fraser, A. Zeller, and A. Orso, "Isolating failure causes through test case generation," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 309-319.

[33] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 225-236.

[34] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010, pp. 317-331.

[35] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 571-572.

[36] K. Sen, D. Marinov, and G. Agha, *CUTE: a concolic unit testing engine for C* vol. 30: ACM, 2005.

[37] J. Shewmaker, "Analyzing dll injection," *GSM Presentation,* 2006.

[38] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang*, et al.*, "BitBlaze: A new approach to computer security via binary analysis," in *Information systems security*, ed: Springer, 2008, pp. 1-25.

[39] M. Staats and C. Păsăreanu, "Parallel symbolic execution for structural test generation," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 183-194.

[40] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*: Pearson Education, 2007.

[41] J. Vanegue, S. Heelan, and R. Rolles, "SMT Solvers in Software Security," in *WOOT*, 2012, pp. 85-96.

[42] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010, pp. 497-512.

[43] T. Wang, T. Wei, G. Gu, and W. Zou, "Checksum-aware fuzzing combined

with dynamic taint analysis and symbolic execution," *ACM Transactions on Information and System Security (TISSEC),* vol. 14, p. 15, 2011.

[44] T. Wang, T. Wei, Z. Lin, and W. Zou, "IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution," in *NDSS*, 2009.

[45] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 511-522.

[46] R.-G. Xu, P. Godefroid, and R. Majumdar, "Testing for buffer overflows with length abstraction," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 27-38.

[47] 黃世昆, 黃銘祥, 黃博彥, 賴俊維, and 呂翰霖, "自動脅迫產生器發展現況與威脅分析," *資訊安全通訊,* vol. 18, pp. 88-100, 2012.

[48] 劉歡, "跨平台 Web 程式測試與攻擊產生系統," 碩士, 資訊科學與工程研究所, 國立交通大學, 2013.