Automated Generating CTF Challenges with Program Analysis Tools Resistance

by

Yue Yin

(Under the Direction of Kang Li)

Abstract

Capture the Flag(CTF) games and competitions have become popular today. Good CTF exercises are not only good approach for learning Cyber Security concepts and techniques in fun, but also useful benchmarks to measure all kinds of popular vulnerability discovering, exploiting, patching techniques or tools. Harder challenges require more sophisticated techniques or tools. On the other hand, the appearance of all the good program analysis tools proposes new challenges for future CTF challenge designers. With the fact that CTF challenges and program analysis tools can complement each other and help each other forward. In this dissertation, we introduce our design method for automated generating CTF challenges. In addition, we propose a new dimension as CTF challenge complexities, which is the ability to resist or obstacle the running of program analysis tools, or referred to as "Resistance to Program Analysis Tools".

Index words: Capture-the-Flag(CTF), Software Vulnerability, Fuzzing, Debugging.

Automated Generating CTF Challenges with Program Analysis Tools Resistance

by

Yue Yin

B.A., Zhejiang University, China, 2012

A Dissertation Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2018

Automated Generating CTF Challenges with Program Analysis Tools Resistance

by

Yue Yin

Major Professor: Kang Li

Committee: Kyu Hyung Lee
Lakshmish Ramaswamy

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
December 2018

Acknowledgments

I dedicate this dissertation to my father, Yuliang Yin and my mother, Meihua Li for everything they did for me. Thanks to all my family and friends for their continual encouragement, support, and advice.

Big thanks to Dr. Kang Li for his mentorship and guidance, without which this work would not have been possible. I would also like to thank to Dr. Kyu Hyung Lee and Dr. Lakshmish Ramaswamy as well as members of UGA NSS Lab for their help.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Capture the Flag(CTF) games and competitions have become popular today. Good CTF exercises are not only good approach for learning Cyber Security concepts and techniques in fun, but also useful benchmarks to measure all kinds of popular vulnerability discovering, exploiting, patching techniques or tools. Harder challenges require more sophisticated techniques or tools. Since automated program analysis techniques such as fuzzing and symbolic execution has been proposed and wildly applied, good ground-truth corpora with vulnerability is required for comprehensive evaluation on those tools. The CTF challenges have been exclusive to human competitors until DARPA sponsored the Cyber Grand Challenges(CGC), a CTF competition to showcase the current automatic program analysis techniques. DARPA's CGC addresses the need for more datasets and evaluation.

On the other hand, the appearance of all the good program analysis tools proposes new challenges for future CTF challenge designers. Firstly, most of the CTF generation works by far are expensive. An automated, systematic method is necessary to meet the high demand of benchmarks from program analysis tools. What's more, new CTF challenges should be able to add obstacles to these techniques and tools, so as to put forward better methods and solutions.

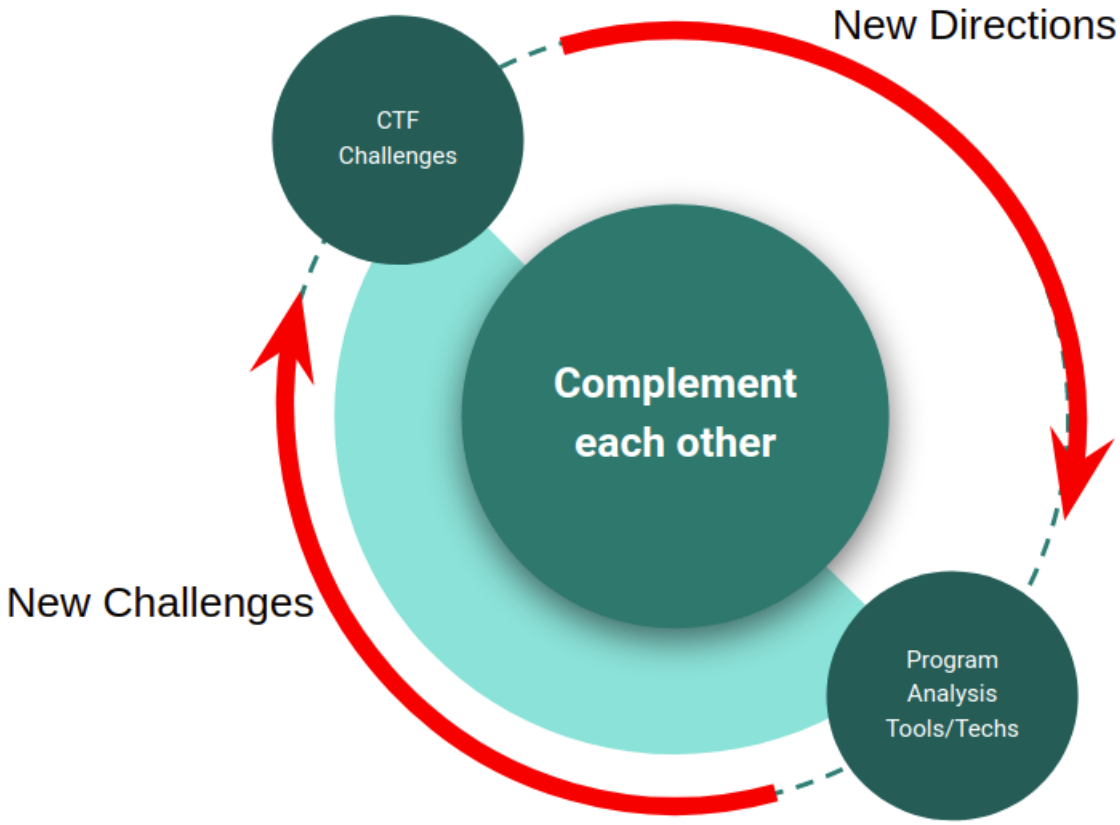


Figure 1.1: CTF challenges and Program Analysis Tools complement each other to getting better

1.2 Research Problems

With the fact that CTF challenges and program analysis tools can complement each other and help each other forward, in this dissertation, we plan to put forward the efficiency improvement on both side.

The program analysis tools we focused on are:

- American Fuzzy Lop(AFL), one of the most successful greybox coverage based fuzzers.
- GNU Debugger(GDB), a useful tool when debugging a program.

And we focused on three different research challenges:

1. After the success of AFL, a lot of improvement solutions has been proposed. Each of them claimed outperforming AFL and other techniques by providing their own evaluation on fuzzing results. Our questions here are:
   (a) Do current fuzzer evaluation works reliable?
   (b) How can we achieve a more comprehensive, more accurate evaluation?
2. AFL's strategies trade off result accuracy with running performance. On of the notable problem brought by those design decisions is a potential "interesting" testcase that AFL just mutated may not be synchronized into AFL's queue directory because of the hash conflicts in coverage bitmap. This brings us the questions:
   (a) How to explain the situation when an "interesting" testcase cannot be synchronized be AFL?
   (b) What can we achieve by taking advantages of AFL's bitmap hash conflict?
3. The appearance of DARPA's CGC addresses the need for more CTF challenges as datasets for evaluation. Such demand requires good solutions on automated generating CTF challenges. The questions here are:

(a) Can we find a good way to automated generate CTF challenges?

(b) What should we emphasize when the CTF challenge consumers are program analysis tools such as AFL or GDB?

1.3 Dissertation Contributions and Roadmaps

Aiming at improving both program vulnerability analysis tools as well as CTF challenge generation, in this dissertation we present our findings on AFL and GDB design imperfections, and introduce new CTF challenge design thoughts based on the findings: a framework that can automated generate CTF challenges in company with a new dimension as CTF challenge complexities,which is the ability to resist or obstacle the running of program analysis tools, or referred to as "Resistance to Program Analysis Tools".

By answering the research problems above, we emphasize the relations of complement among CTF challenges, Fuzzing techniques(such as AFL) and Debugging Tools(such as GDB) in Fig. 1.2:

1. Good CTF challenges can be used as ground-truth benchmark to evaluate fuzzers.

2. On the other hand, fuzzing techniques put forward the improvements in the design of CTF challenges

3. The gdb vulnerabilities found by fuzzers can be exploited in adding the feature "Resistance to Debugging Tools" to CTF challenges.

The remainder of the dissertation is organized as follows:

After Literature Review(Chapter 2), we first introduce introduce a new CTF challenge design style SRCTF as well asa our Fuzzing-Benchmark design to make fuzzer evaluation work more systematic and comprehensive(Chapter 3). Then, we present Deafl, a binary injection tool to defeat AFL fuzzing by exploiting AFL's bitmap hash conflict(Chapter 4). Last, we present different scenarios of defeating GDB with malformed ELF metadata in program,

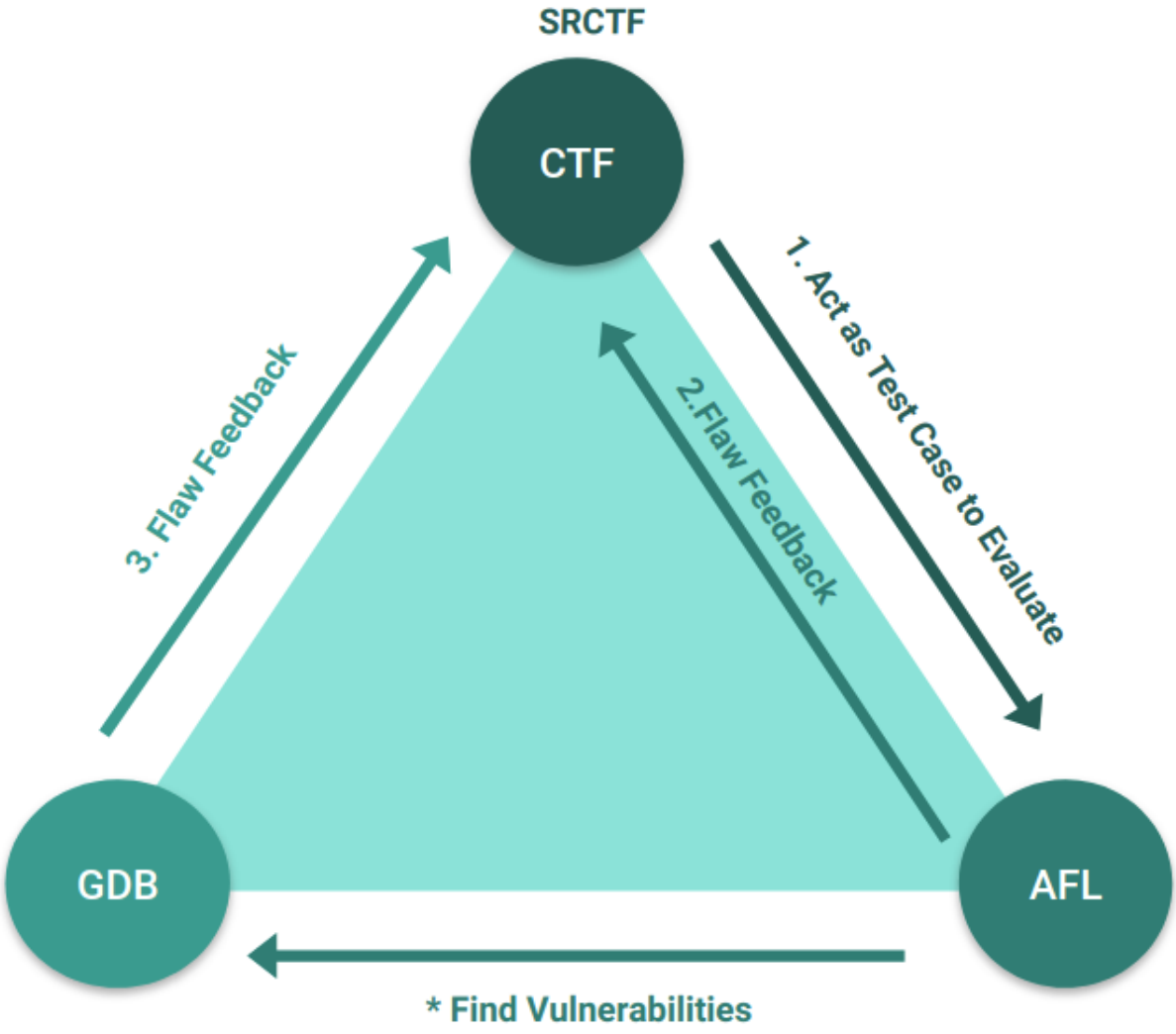


Figure 1.2: The relations among CTF challenge, AFL and GDB.

which leads to a new dimension for CTF challenges: "Resistance to Program Analysis Tools" (Chapter 5).

Chapter 2

Literature Review

2.1 Program Analysis Tools

2.1.1 Fuzzing and American Fuzzy Lop(AFL)

Fuzzing techniques has been widely-used as an effective software testing technique to find program vulnerabilities [20, 33]. It was originally invented as a testing method that cause a target program to crash by generating random inputs. State-of-the-art fuzzers are usually mutation-based, which mutate existing testcases to get new input. According to method of input mutation, there are three types of fuzzers: whitebox, blackbox and gerybox. While whitebox fuzzer requires source code [13, 12, 14] and blackbox fuzzer has no information to target program[28], greybox fuzzers leverage light-weight analysis tools on binary code to get the knowledge of target program.

American Fuzzy Lop(AFL) [37] is one of the most successful fuzzing tools. It can be used as whitebox fuzzer when source code is available and can also act as greybox fuzzer by implementing light-weight instrumentation to binary file. For example, with the help of

Table 2.1: Different types of fuzzer.

Technique	**Source Code?**	**Lightweight Analysis?**
Blackbox	No	No
Whitebox	Yes	No
Greybox	No	Yes

QEMU user emulation mode [3], AFL can get basic block information similar to compiler-instrumented code.

AFL also applies generic algorithms to generate new testcases. Specifically, the mutation strategies are:

1. Deterministic strategies
   (a) Bit flips with varying lengths and stepovers, including byte flips.
   (b) Addition and subtraction of small integers.
   (c) Insertion of known interesting integers(0, 1, INT_MAX, .etc).
2. Non-deterministic strategies:
   (a) Havoc: Stacked bit flips, insertions, deletions, arithmetics
   (b) Splice: Splicing two distinct input files at a random location

In order to explore different paths of the target program, AFL examine newly generated inputs based on branch coverage information, which is known as "bitmap". It is a hash table AFL keeps to record the times executed for each branch. For each new execution's bitmap, AFL compares the specific path bitmap with its global bitmap which contains all branches covered before and will identify such input "interesting" if the comparison shows differences.

With the help of AFL's good designs, countless vulnerablities on notable programs as well as all sorts of less-widespread software have been found [37]. Unfortunately, AFL's strategy suffers many limitations. One of the most obvious shortages of AFL is it usually fails to generate values for branches with context dependent data or complex condition constraints.

In order to increase the efficiency of fuzzing work, people add heuristics or other analysis information to AFL and proposed many AFL-like tools [4, 5, 18, 19, 30, 32]. AFL also introduces dictionary mode, which improves ability on solving constraints like "magic bytes". All of these fuzzers trade off strategy accuracy with running performance because it is infeasible

to track all path coverage information for practical larger programs. Managing large path coverage information will always result in colliding information or conflicts.

Among all the new implementations, FAIRFUZz is an interesting open-source extension of AFL designed by Caroline Lemieux et al. from UC Berkeley [18]. It just adds around 600 lines of C code to AFL's core implementation for mainly two key steps:

- FAIRFUZZ can prioritize inputs exercising rare parts of the program under test.
- FAIRFUZZ adjust the mutation of input part with respect to rare parts of the program.

According to their evaluation, FAIRFUZZ outperforms other tools such as AFL, AFLFast.

2.1.2 Debugging and GNU Project Debugger(GDB)

A debugger is a tool to give user a view of the running program in a natural and understandable way, and provide control over the execution as well as useful information. With the help of debuggers, we can interact with and examine the state of running program by setting breakpoints and examining memory and register contents. It requires the debugger coordinating with the compiler, which converts human-readable source code into machine language, to make a program debuggable. The GNU Debugger (gdb) is the standard debugger for most Linux systems. It allows you to see what is going on "inside" another program while it executes, or what another program was doing at the moment it crashed.

DWARF (debugging with attributed record formats) [11] is a debugging file format used by many compilers and debuggers to support source-level debugging. It is the format of debugging information within an object file. The DWARF description of a program is a tree structure where each node can have children or siblings. The nodes might represent types, variables, or functions.

DWARF uses a series of debugging information entries (DIEs) to define a low-level representation of a source program. Each debugging information entry consists of an identifying tag and a series of attributes. An entry or group of entries together, provides a description of

a corresponding entity in the source program. The tag specifies the class to which an entry belongs and the attributes define the specific characteristics of the entry.

2.2 CTF challenges/Testcase generations

Capture the Flag (CTF) is a computer security competition. This kind of contests are usually designed to serve as an educational exercise to give participants experience in securing a machine, as well as conducting and reacting to the sort of attacks found in the real world. Reverse-engineering, network sniffing, protocol analysis, system administration, programming, and cryptanalysis are all skills which have been required in prior competitions. CTFs are touted as powerful education and training vehicles [2, 6, 36].

2.2.1 Introduction of CTF

There are two main styles of capture the flag competitions: attack/defense and jeopardy. In attack-defend CTFs, teams run services on a shared network and compete to compromise or disrupt other's services while keeping their own services available. In jeopardy-style CTFs, teams solve puzzle-like challenges in order to score points. Jeopardy-style CTFs have many categories, there are common types such as:

- Pwnables

  Intentionally vulnerable programs that can be exploited to obtain a flag.

- Reverse Engineering

  Obfuscated programs that must be reverse engineered to reveal a flag.

- Crypto

  Weak or poorly implemented cryptography;generally the flag is hidden in an encrypted message that must be decrypted.

- Web

  A web site with some combination of vulnerabilities (e.g. SQL injection and/or cross site scripting)that can be exploited to reveal a flag.

In this dissertation, by stating "CTF challenges", we mean jeopardy-style challenges in the format of ELF binary program, known as the category "pwnable" or "PWN".

2.2.2 Vulnerable Program as Test Data

CGC and LAVA are two notable works on designing "PWN" challenges and they are both the popular benchmarks for automated analysis tools such as fuzzing and symbolic execution [16].

- CGC

  Automated software vulnerability analysis used to be a very difficult—and generally unsolvable—problem. Staring from June 2014, DARPA launched the Cyber Grand Challenge (CGC), a competition designed to spur innovation in fully automated software vulnerability analysis and repair. Too often, computer science researchers don't provide enough information or access to test data so as to allow verification and validation of results or comparison of competing approaches. DARPA has provided a platform to address these issues[31].

  On August 4, 2016, the Defense Advanced Research Projects Agency(DARPA) held a Cyber Grand Challenge(CGC), a competition to create automatic defensive systems capable of reasoning about flaws, formulating patches and deploying them on a network in real time [9]. DARPA releases the binaries used in the event of CGC, which has become another popular benchmark for evaluation. However, these binaries need to be run under the DARPA Experimental Cyber Research Evaluation Environment(DECREE), which requires more efforts to deploy analysis tools.

- LAVA

Dolan-Gavitt et al. [10] present LAVA in order to fill the shortage of ground-truth corpora for automating vulnerability discovery. They use LAVA corpora to evaluate the detection ability for automated bug finding tools. The paper reports results by running FUZZER, symbolic execution and S2E [7] approaches on their testing corpora **LAVA-M**. Since then, **LAVA-M** has been widely use as the evaluation metrics for many bug finding techniques such as [5, 30].

Only CGC and LAVA can not match the large demands of evaluating tools. There are other notables works also trying to underlines the importance of vulnerable test data [22, 29].

In EvilCoder [29], they automatic generated bug-ridden test corpora by modifying security checks in source code statically. This method requires extra effort to find crash-triggered input after the generation.

We also find [26] on the website of National Institute of Standards and Technology(NIST). It was designed as another ground-truth corpora to test techniques on different types of CWE. However, this test bundle is specifically for testing static analysis tools.

In [1], the authors proposed 3 characteristics for vulnerability testcases after evaluating some static analysers.

- Statistical significance:

  Means many, diverse vulnerabilities.

- Ground truth:

  The location of the vulnerabilities must be known.

- Relevance:

  The vulnerabilities should represent practical problems found in real program.

Test corpora for evaluation should fulfill these lines.

Chapter 3

Automatically Generating CTF challenges for Fuzzing Tools Evaluation

3.1 Introduction

Capture the Flag(CTF) competitions have become popular today. Good CTF exercises are not only good approaches for learning cyber security concepts and technical in fun, but also good tools to measure all kinds of popular vulnerability discovering, exploiting, patching technologies. According to popular CTF website CTFtime, usually there were more than one hundred CTF events a year(141 in 2017, 109 in 2016). Participating in multiple events can bring good practice for both students and security experts. However, with the growing trend of automated vulnerability detection techniques used to solve CTF challenges, the need of more and better CTF challenges is urgent. Such challenges should not only keep all typical CTF characteristics but also require more state-of-the-art program analysis tools to solve. The large demand for more and good CTF challenges poses a new challenge to CTF designers because most of the CTF generation work are expensive.

State-of-the-art fuzzers usually take LAVA or other popular testcases as ground-truth corpora for comparison and evaluation. By stating having most bugs found in the same program, one tool can claim it outperforms other fuzzers. However, most of the evaluation works ignore the fact that fuzzing/testing result can be affected by multiple factors/dimensions.

Research Goals: In this chapter, we aim to answer the following research questions:

- What should we care when generating CTF challenges?
- Do current fuzzing tools evaluation works reliable? How can we achieve a more comprehensive, more accurate evaluation?

To answer these questions, we first propose "SRCTF", a novel framework for automated generating CTF challenges that are both "Stepwise" and "Reusable". We then present case studies to show that most of the previous evaluation ignored some key factors that may affect the result. Failing to provide enough details about experiments make the evaluation less reliable. In order to answer the second question, we proposed our list of necessary metrics as well as a series of programs as testing benchmark. The goal of this work is to make evaluation work on program analysis tools more accurate and reliable.

3.2 Related Work

3.2.1 LAVA and Autoctf

Dolan-Gavitt et al. [10] present LAVA in order to fill the shortage of ground-truth corpora for automating vulnerability discovery. They use LAVA corpora to evaluate the detection ability for automated bug finding tools. The paper reports results by running FUZZER, symbolic execution and S2E approaches on their testing corpora **LAVA-M**. Since then, **LAVA-M** has been widely use as the evaluation metrics for many bug finding techniques such as [5, 30].

With LAVA's vulnerability injection technique, the same group designs new CTF challenges and creates AutoCTF, a week long CTF event [15]. This work makes CTF designs cheap and reusable.

3.2.2 LAVA: Automated Bug Insertion

Dolan-Gavitt et al. [10] present LAVA in order to fill the shortage of ground-truth corpora for automating vulnerability discovery. They use LAVA corpora to evaluate the detection ability for automated bug finding tools. The paper reports results by running FUZZER, symbolic execution and S2E approaches on their testing corpora **LAVA-M**. Since then, **LAVA-M** has been widely use as the evaluation metrics for many bug finding techniques such as [5, 30].

```
unsigned int lava_get(unsigned int bug_num) {
#define SWAP_UINT32(x) (((x) >> 24) | (((x) & 0x00FF0000) >> 8)
   | (((x) & 0x0000FF00) << 8) | ((x) << 24))
    if (0x6c617661 - bug_num == lava_val[bug_num] ||
        SWAP_UINT32(0x6c617661 - bug_num) == lava_val[bug_num])
           {
        dprintf(1, "Successfully triggered bug %d, crashing
          now!\n", bug_num);
    }
```

Figure 3.1: The lava_get() function defined to trigger crash signal.

```
while (!feof (stdin))
  {
    char *thisfield;
    size_t thislen;
    bool new_group;

    if (readlinebuffer_delim (thisline+(lava_get(215))*(0x6c61758a==(lava_get(215))||
      0x8a75616c==(lava_get(215))), stdin, delimiter) == 0)
      break;
```

Figure 3.2: An example of LAVA's code insertion into binutils uniq.c source code.

LAVA has been widely used as ground-truth corpora for evaluation fuzzing tools. However, it focus only on adding conditional path with magic bytes constraints. An example of program source code modified by LAVA is shown as Fig. 3.1 and Fig. 3.2. Such insertion reveals little real world vulnerability schemas. Additionally, our testing on **LAVA-M** shows that the magic bytes can be collected by static analysis and an AFL instance with a dictionary including such magic bytes can easily find most of the bugs they mined in a short time.

3.2.3 CGC

Automated software vulnerability analysis used to be a very difficult—and generally unsolvable—problem. Staring from June 2014, DARPA launched the Cyber Grand Challenge (CGC), a competition designed to spur innovation in fully automated software vulnerability analysis and repair. Too often, computer science researchers don't provide enough information or access to test data so as to allow verification and validation of results or comparison of competing approaches. DARPA has provided a platform to address these issues[31].

On August 4, 2016, the Defense Advanced Research Projects Agency(DARPA) held a Cyber Grand Challenge(CGC), a competition to create automatic defensive systems capable of reasoning about flaws, formulating patches and deploying them on a network in real time [9]. DARPA releases the binaries used in the event of CGC, which has become another popular benchmark for evaluation. However, these binaries need to be run under the DARPA Experimental Cyber Research Evaluation Environment(DECREE), which requires more efforts to deploy analysis tools.

3.2.4 NIST Test Suites

We find [26] on the website of National Institute of Standards and Technology(NIST). It was designed as another ground-truth corpora to test techniques on different types of CWE. However, this test bundle is specifically for testing static analysis tools.

3.2.5 FairFuzz

Among all the new implementations, FairFuzz is an interesting open-source extension of AFL designed by Caroline Lemieux et al. from UC Berkeley [18]. It just adds around 600 lines of C code to AFL's core implementation for mainly two key steps:

- FairFuzz can prioritize inputs exercising rare parts of the program under test.
- FairFuzz adjusts the mutation of input part with respect to rare parts of the program.

According to their evaluation, FairFuzz outperforms other tools such as AFL, AFLFast. In this chapter, we will set up experiments for a comprehensive evaluation including AFL and FairFuzz.

3.3 SRCTF: Generating Stepwise and Reusable CTF

As we have discussed, today's CTF challenges have been entrusted the responsibilities of:

- Spread security techniques.
- Measure security skills.
- Strengthen technical and management skills.

3.3.1 What is a Good CTF Challenge

The cost of designing and creating CTF challenges is usually high. Challenges are often created by expert volunteers with large amount of time, the difficulty level cannot be . Moreover, after the write-ups of challenges released, challenges can rarely be used again.

Since most of the current CTF challenges are not well formalized and systematic, and a good CTF challenge is always prone to lose most of its value once the solution released. We propose the following two important characteristics for CTF challenges:

- Stepwise:

  A user's skillset should "grow" as they progress from one challenge to the next. That growth should be relied upon for later challenges. A natural progression from challenge to challenge should be evident.

- Reusable:

  Each challenge should be "tunable" such that unique variants can be generated with minimal effort, a newly generated variant should has the same difficulty level as the original one.

Table 3.1: Stack Buffer Overflow Key Factors

Factor	**Stepwise?**	Note	**Reusable?**	Note
Buffer Size	Y	(The less the harder)	Y	(In proper range)
Buffer Variable Location	Y	(NA)	Y	(NA)
Accept Input Length	Y	(The less the harder)	Y	(In proper range)

3.3.2 SRCTF Design

We first introduce our new Stepwise and reusable CTF framework SRCTF, with the goal of enhancing and broadening computer security education through the development of "Stepwise" and "Reusable" CTF challenges that emphasize problem solving. 5.3.2.1 Approach For each kind of vulnerability, we design a series of challenges with different difficulty level. To achieve such difficulty level chain, we investigate and decide the challenge key factors based on the following three categories: Environment settings: Programs are sensitive to environment settings of host OS, enable or disable one may turn into a total different(in difficulty level/vulnerability type/attack method) challenge. Vulnerability-related: Each vulnerability has target factors related to the vulnerability topic, such factors can be used to tune difficulty level. Factors for specific task(method/skill related): There is a big gap between identifying a bug and actually exploiting it. Such gap usually requires advanced methods and skills to attack. Some factors during those methods or skills can be used to apply stepwise/reusable designing.

Base on findings in Table. 3.1, given "Stack Buffer Overflow" as vulnerability type, we have a 6-level difficulty chain with setup as Fig. 3.3.

By analysing factors list in [table], we can also find the tunable factors which make challenges reusable. For "Stack Buffer Overflow", "the size of buffer" and "the size of accept input" are two of the most important factors, by assigning them appropriately, we can get a

- Type: Stack Buffer Overflow
- Environment: Linux, x86
- Compile Configs:
  - -m32 32-bit
  - -z noexecstack NX(Non-executable stack)
  - -fno-stack-protector No Canary(stackprotector)
- Expected Solution: Code Reuse After Exploitation
- Knowledge Difficulty Chain:
  - Level 0: (For demonstration)
  - Level 1: Overwrite flag value
  - Level 2: Overwrite return address to call help function
  - Level 3: Overwrite return address to call help function + Passing parameters
  - Level 4: Use “pop-ret” to chain two help functions
  - Level 5: Build own ROP chain
  - Level 6: Input has character restriction(Ascii Armoring)

Figure 3.3: Stack Buffer Overflow Setups

set of variants share the same difficulty level. The only thing it affects is the attack payload size.

With the design thoughts to achieve "stepwise" and "reusable" goals. The actual implementation formula we used is:

$$Challenge = ProgramTemplate + BuggyCodeSnippets + NecessaryHelpFunctions$$

To make the challenge more realistic, the program templates usually are actual small program with source-code available, for example a simple game or a tool that consuming string data such as checksum, md5 and so on. The buggy code snippet contains the lines of code that triggered the vulnerability as well as the tunable factors which will be assigned when assembling the program. Take topic "Stack Buffer Overflow" for instance, we wrote a pure game program ropeman(same as hangman) as the template and picking the a function handling menu selection as the buggy code snippet.

The buggy code here is selectContinue(), and key factors BUF_SIZE, MAX_SIZE is assigned when generating the challenge, as Fig. 3.6 shows.

So that each user gets a different challenge with same difficulty level.

In Fall Semester 2016, we used SRCTF as a Practice System for UGA computer science course: CSCI4250/6250-Computer Security, which has about 40 undergraduate and graduate students. The system was hosted on Ubuntu 14.04 with a Python Django Web Application+Mysql, each user is assigned a docker container to provide all challenges and necessary environment setting. As the course proceed, we open more challenges that covers many topics:

1. Pwn
    (a) Stack Buffer Overflow
    (b) Format String
    (c) Double Free

```
int read_flag(){
    system("cat ../rw/flag");
    return 0;
}

int selectContinue() {
    char sSelection[BUF_SIZE]   = {0};
    int rtnCode                 = 0;

    printContinue2();

    //this is where the overflow occurs
    getStr(sSelection, MAX_SIZE, stdin);
    switch (sSelection[0]) {
        case 'S':
        case 's':
            rtnCode = 1;
            break;
        case 'X':                       // quit
        case 'x':
            rtnCode = 2;
            break;
        default:
            output("Please enter a valid command!(S | X)\n  ");
            break;
    }

    return rtnCode;
}
```

Figure 3.4: Buggy Code Snippets + Necessary Help Functions for "ropeman" level 2

```
char* not_used = "cat ../rw/flag";

int read_flag(){
    system("echo ../rw/flag");
    return 0;
}

int selectContinue() {
    char sSelection[BUF_SIZE]   = {0};
    int rtnCode                 = 0;

    printContinue2();

    //this is where the overflow occurs
    getStr(sSelection, MAX_SIZE, stdin);
    switch (sSelection[0]) {
        case 'S':
        case 's':
            rtnCode = 1;
            break;
        case 'X':                       // quit
        case 'x':
            rtnCode = 2;
            break;
        default:
            output("Please enter a valid command!(S | X)\n:  ");
            break;
    }

    return rtnCode;
}
```

Figure 3.5: Buggy Code Snippets + Necessary Help Functions for "ropeman" level 3

```
echo \#define BUF_SIZE BS > config.h

rand_bsize="$((RANDOM%(8370000-1+1)+1))"
 #define MAX_SIZE        BUF_SIZE + 40
```

Figure 3.6: Configs setting for "ropeman"

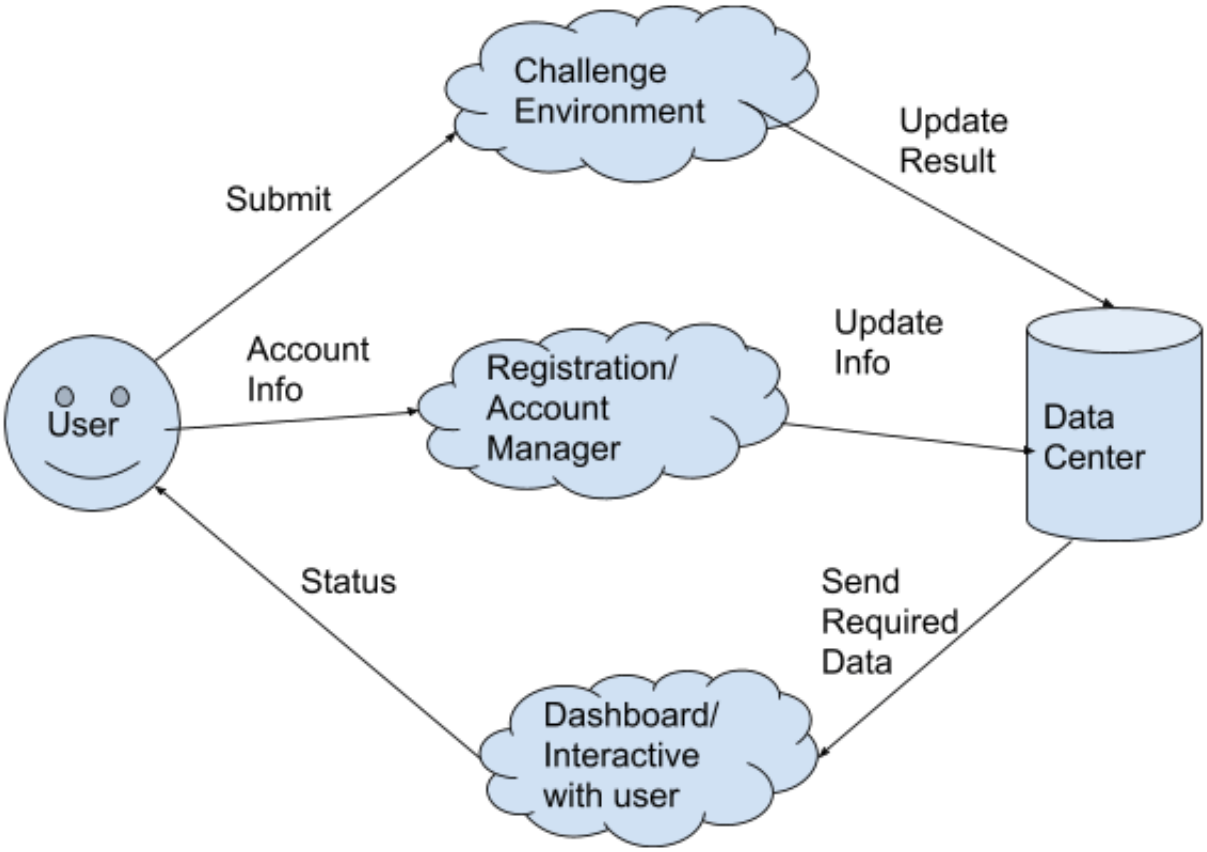


Figure 3.7: SRCTF system implementation

2. Reverse:

   (a) ELF series

   (b) Android Apps series

3. Web:

   (a) SQL Injection

   (b) Trivia(Webpage comments, Path traverse, Cookies .etc)

The feedbacks show that our "Pwn" challenges are probably too hard for most of new players. But students can understand the related security knowledge as well as the difficulty chain after release the solution. In addition, serving challenges to 40 students with no same challenges shows the effectiveness of reusable design, every student can strengthen related cyber security skills by solving own unique challenges.

We believe that the hands-on experience, both in workshops and actual competitions,is beneficial in invoking interest and teaching the CyberSecurity subject to students. The SRCTF project is also released on github: http://tunablectf.com.

3.4 Motivation: Using CTF Challenges for Better Fuzzing Tools Evaluation

Most state-of-the-art fuzzers take LAVA or CGC samples or other popular testcases as ground-truth corpora for comparison and evaluation. By stating having most bugs found in the same program, one tool can claim it outperforms other fuzzers. However, most of the evaluation works ignore the fact that fuzzing/testing result can be affected by multiple factors/dimensions. In [17], It shows most of the recent fuzzing papers failed to follow a good methodology for evaluation work. To emphasize the possible consequence of bad evaluation, we have conducted some experiments.

```
int DBZ(uchar *out[]){
    int a = 0;
    int ret = 0;

    if(strcmp(out[0], "Must") == 0){
        if(strcmp(out[1],"Area") == 0){
            if(strcmp(out[2], "Geek") == 0){
                if(strcmp(out[3], "Inch") == 0){
                    if(out[4][0] == 'C'){
                        a = 10/(out[5][0]-'!');
                        ret = 5;
                    }
                    else{
                        ret = 4;
                    }
```

Figure 3.8: A test case that contains a divide-by-zero vulnerability

3.4.1 Factors that Can Affect Fuzzing Results

1. **Input Seed**

   Most of the previous work's evaluation part did not reveal any detailed references of the input seed used. One fact that we cannot ignore is fuzzers running result is sensitive to the seed input provided. To illustrate this fact, we provide a simple experiment.

   Experiment:

   We write a simple program as the test target, which has a divide-by-zero vulnerbility as Fig. 3.8.

   The program will take input from STDIN and process it with a series of format check path constraints. To trigger the divide-by-zero vulnerability, the input should pass a few "magic bytes" checking and two more "char" checking:

```
out[0]    = "Must"
out[1]    = "Area"
out[2]    = "Geek"
out[3]    = "Inch"
out[4][0] = 'C'
out[5][0] = '!'
```

The magic bytes constraints information can be achieved by simple static analysis, for example, using command "strings" as Fig. 3.9. By collecting this information as a dictionary, AFL dictionary mode works better in this case. Similar methods can also be used to solve LAVA challenges.

We analysis two fuzzers by given two different seed input.

- Tool 1: AFL 2.52b with dictionary
- Tool 2: AFL 2.52b with more byte flips strategies modified by us
- Seed 1: "a"
- Seed 2: "MustAreaGeekInchXXX"

With the help of dictionary as Fig. 3.10, Tool 1 should be able to pass the first four "magic bytes" checking easily and solve the last two "char" checkings in rest of times. At the same time, Tool 2 would make no progress on this challenge for most of the input seed(e.g. Seed 1), since AFL is famous for doing poorly on "magic bytes" path constraints. However, in some extreme cases, if the given input seed is Seed 2, which contains enough information for "magic bytes", the result of running Tool 2 would be different.

Result:

- Running with Seed 1:

  We run Tool 1 with Seed 1 3 times, all three tries can find crash within 20 minutes.

  While at the same time, Tool 2 cannot proceed any more when it reaches the first "magic byte" condition.

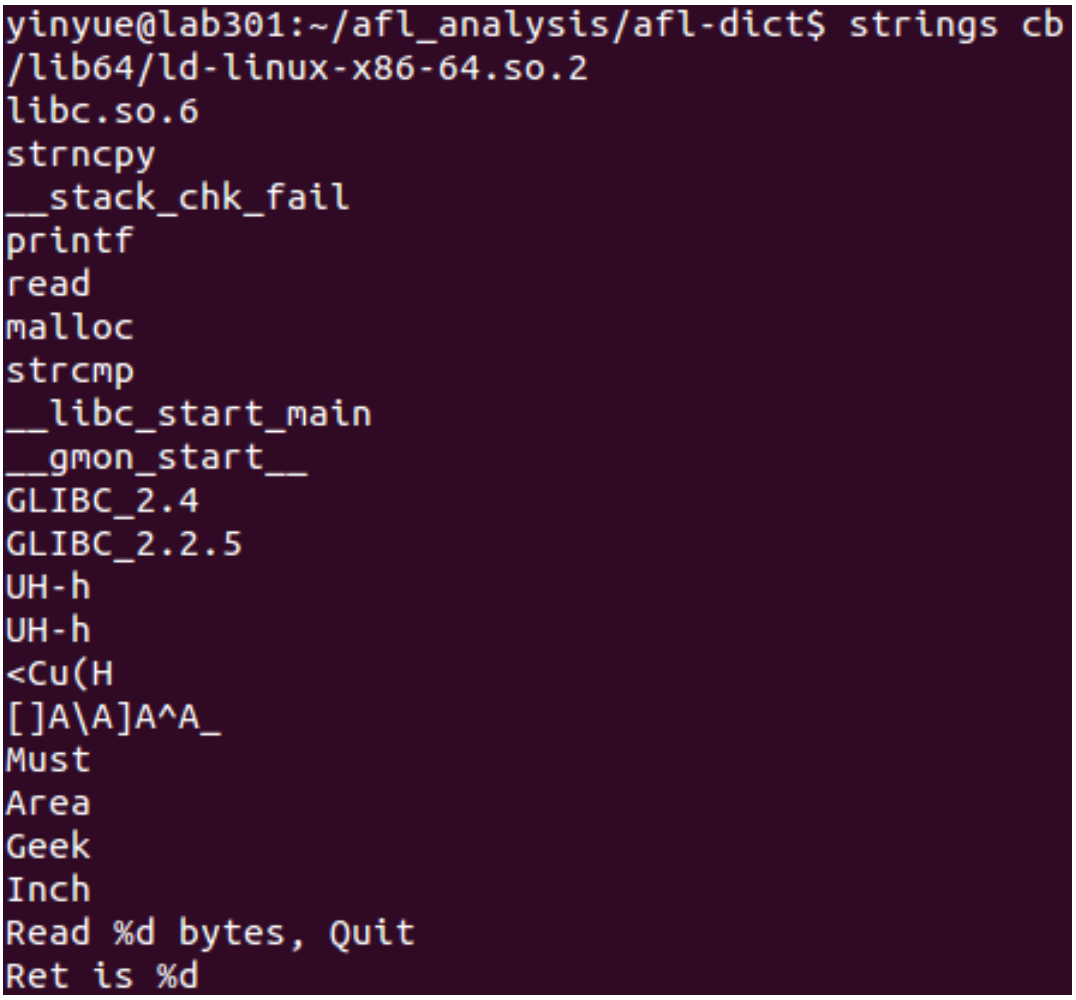


Figure 3.9: Command "strings" result for test binary

```
1  keyword_area="Area"
2  keyword_arguments="arguments"
3  keyword_break="break"
4  keyword_case="case"
5  keyword_catch="catch"
6  keyword_const="const"
7  keyword_continue="continue"
8  keyword_debugger="debugger"
9  keyword_decodeURI="decodeURI"
10 keyword_geek="Geek"
11 keyword_inch="Inch"
12 keyword_must="Must"
13 keyword_magic="MAGIC"
```

Figure 3.10: A simple dictionary for Tool 1

- Running with Seed 2: When fuzzing using Seed 2, Tool 1 show similar running results. There's one running shows Tool 1 cannot find the crash in 40 minutes.

  When given Seed 2, which has "magic bytes" information, Tool 2 can explore all paths in target binary. Moreover, because of our modified strategy on "Byte Flips", it shows better results than Tool 1. By performing more "Byte Flips" in deterministic process, Tool 2 can get higher chance to solve 1-byte-char constraints, instead of leave the solving to non-deterministic process, which has many uncertainty and randomness.

Summary:

In summary, Tool 1 is better than Tool 2 with Seed 1 or most of the seeds. However, Tool 2 outperforms Tool 1 when the given input is seed 2. Though it is an extreme case, the fact that seed input can influence the fuzzing result cannot be ignores.

When evaluating different fuzzers, we suggest researchers to list the details of seed input and make sure the seed is not biased to any testers.

2. **Bug Types**

   Most of the previous evaluation on fuzzing tools focused only on "number of bugs found" but not mention the types of bugs. We believe different tools may have different coverage on "workable types of bugs"

   Experiment:

   We pick 5 types of bug that occurred a lot in C program language:

   - CWE-89 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
   - CWE-120 Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
   - CWE-862 Missing Authorization
   - CWE-134 Uncontrolled Format String
   - CWE-122 Heap-based Buffer Overflow
   - CWE-365 Divide-by-Zero
   - CWE-835 Infinite Loop

   The first four CWEs are listed as "Top 25 Most Dangerous Software Errors", according to [21] released by MITRE,

   To test fuzzing tools' ability on triggering specific types of bug, we select a simple code snippet contains such vulnerability from [26] and put it directly in main() function as test target program.

   We use two tools in this experiments:

   - Tool 1: AFL 2.52b
   - Tool 2: AFL 2.52b with ASAN supported

Table 3.2: Fuzzing tools' ability on triggering specific types of bugs

Tools	CWE89	CWE120	CWE122	CWE134	CWE365	CWE835[1]	CWE862
1	N	Y	N	Y	Y	Y	N
2	N	Y	Y	Y	Y	Y	N

Result:

The running result is in Table 3.2.

Summary:

Taking "number of crashs found" as the evaluation metric ignores the following facts:

(a) Most fuzzers is not suitable for CWEs that not ends in crash.

(b) Some of the CWEs happend in the format of "hang" instead of "crash", for example CWE835.

(c) Different fuzzers may have different CWE triggering coverage results

To make the fuzzing tools evaluation work more reliable, types of triggerable bugs should be taken into consideration.

3. **Other Configurations**

Besides input seed and bug types, other running configurations are also important to running fuzzers. A comprehensive evaluation should also mention using scopes for different fuzzers such as:

- Can be used under which environment(OS), 32-bit or 64-bit.
- Can be used to fuzz library or not.
- Can fuzz when input's format is FILE/STDIN/parameters

[1]This CWE can be triggered as "hangs", not "crashes"

3.5 Design CTF Challenges with Comprehensive Evaluation Metrics

We believe a more comprehensive evaluation method is necessary to fully understand the ability of fuzzers. We recommend to track all of the following metrics, which show different aspects of a fuzzer's running ability:

1. Ability of Exploring Path:
    (a) Block Coverage
    (b) Edge Coverage
    (c) Path Coverage
    (d) Vulnerable Block Coverage
    (e) Useful/All Testcase Effectiveness
2. Ability of Triggering Bugs:
    (a) Number of Crashes
    (b) Number of Unique Crashes
    (c) Unique Crash Ratio
    (d) Crash/Useful Testcase Effectiveness
    (e) Crash/Total Testcase Effectiveness

In addition, we add a checklist to fully understand the fuzzer's using scope: Using Scope (Can or cannot work on/with...):

1. Using Scope(Can or cannot work on/with...):
    (a) Types of CWE that can trigger
    (b) OS/Platform that work on
    (c) Target feature(library, browser, protocol...)

As the table shows, previous works only focus on part of the metric item we listed, which make the evaluation result less reliable. (lava, vuzzer, angora)

3.5.1 Benchmark Metrics

3.4.1.1 [A]Running Capability

This category shows a fuzzer's fuzzing ability by listing numbers on different aspects.

- **Path Exploring[A1]**

  A good fuzzer should have good performance on finding new behaviors in a program. By giving fixing time of fuzzing, a larger(or smaller) number in a metric can prove better performance of such fuzzer tool. More specifically, better fuzzer should have larger(or smaller) number in the following metric:

  [a1] Block Coverage

  Shows the ratio of basic block found by fuzzer. To calculate this metric:

  $$A1 = (number\ of\ basic\ blocks\ covered)/(total\ number\ of\ basic\ blocks\ in\ program)$$

  For instance, in AFL-fuzz, the "number of basic blocks covered" is calculated by analysing all testcase in afl queue directory using QEMU-trace. The "total number of basic blocks in program" is get from the program's CFG.

  [a2] Edge Coverage

  A edge in a program's CFG means a transition between 2 basic blocks. To calculate this metric:

  $$A2 = (number\ of\ edges\ covered)/(total\ number\ of\ edges\ in\ program)$$

  For instance, in AFL-fuzz, the "number of edges covered" is calculated by analysing all testcase in afl queue directory using QEMU-trace. The "total number of edges in program" is get from the program's CFG.

[a3] Path Coverage(Path Number)

The path coverage is unavailable usually, because a program may have infinite number of paths. However, by directly comparing paths found on the same target program, we can tell the differences of fuzzers path finding ability

$$A3 = (number\ of\ paths\ covered)$$

For instance, in AFL-fuzz, the "number of paths covered" is the total number of files in afl's "queue" directory.

[a4] Vulnerable Block Coverage

Vulnerable block is the basic block that contains vulnerability. A higher "vulnerable block coverage" shows a fuzzer is more "sensitive" to vulnerable points when exploring.

$$A4 = (number\ of\ vulnerable\ blocks\ covered)/(number\ of\ basic\ blocks\ covered)$$

For instance, in AFL-fuzz, the "number of vulnerable blocks covered" is the number of unique vulnerable blocks covered by afl queue testcases, the "number of basic blocks covered" is calculated by analysing all testcase in afl queue directory using QEMU-trace.

[a5] Interesting Testcase Effectiveness

A fuzzer with better "Interesting Testcase Effectiveness" will generate interesting test-cases with less number of testcase generation, thus yields better productivity.

$$A5 = (number\ of\ interesting\ testcase)/(total\ number\ of\ testcases\ generated)$$

For instance, in AFL-fuzz, the "number of interesting testcase" is the total number of files in afl output's "queue", "crash: and "hang" directory, the "total number of testcases generated" is the "total execs" number.

- **Triggering Bugs[A2]**

  [a6] Number of Crash

  A fuzzer's general goal is to find as much crash as possible.

$$A6 = (number\ of\ crashes\ found)$$

  For instance, in AFL-fuzz, the "number of crashes found" is the number of "total crashes".

  [a7] Number of Unique Crashes If a single bug can be reached in multiple ways, there will be some count inflation. Counting unique crashes would be a more accurate metric for evaluation. This is also one of the most important metrics that can directly tell if a fuzzer is good or not.

$$A7 = (number\ of\ unique\ crashes\ found)$$

  For instance, in AFL-fuzz, crashes are considered "unique" if the associated execution paths involve any state transitions not seen in previously-recorded faults. The "number of unique crashes found" is the number of "uniq crashes".

  [a8] Unique Crash Ratio

  The unique crash ratio shows a fuzzer's sensitivity of finding crashes in different spot.

$$A8 = (number\ of\ unique\ crashes\ found)/(number\ of\ crashes\ found)$$

  [a9] Crash Input Effectiveness(1)

  The effectiveness of generating crash input from known testcases, with respect to interesting testcases generated.

$$A9 = (number\ of\ unique\ crashes\ found)/(number\ of\ interesting\ testcase)$$

[a10] Crash Input Effectiveness(2)

The effectiveness of generating crash input from known testcases, with respect to total testcases generated, in other words, the time of time that the target program executed.

$$A10 = (number\ of\ unique\ crashes\ found)/(total\ number\ of\ testcases\ generated)$$

3.4.1.2 [B]Using Scope

This category shows a fuzzer's suitable usability by listing a series of checklist. Including:

- **[b1] Type of Crashes**

  We list the common vulnerabilities by their CWEs. Such as CWE-120(Buffer Copy without Checking Size of Input), CWE-134(Uncontrolled Format String), CWE-122(Heap-based Buffer Overflow) and so on.

- **[b2] OS/Platform**

  We also care on which platform we can use a fuzzer, such as "Linux", "MacOS", "Windows"

- **[b3] Specific Target**

  Some fuzzers are designed for specific targets, such as "Browser Fuzzer", "Network Protocol Fuzzer", "Cloud Fuzzer", "Virtual Machine Fuzzer", "Library Fuzzer".

3.5.2 Evaluation Metrics Used in Previous Work

We analyse evaluation methods and metrics used in previous work and summarize the information in Table 3.3.

As we can see, most of the previous work list the number of bugs found as well as few more metrics.

Table 3.3: Evaluation Metrics Used in Previous Work.

Works	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**A10**	**B1**	**B2**	**B3**
LAVA [10]						Y							
Angora [5]		Y			Y	Y				Y			
VUzzer [30]						Y				Y			
FairFuzz [18]		Y				Y				Y			
AFLFast [4]		Y			Y	Y					Y		
Steelix [19]		Y				Y					Y		

Table 3.4: Tools for experiments.

Tools	**Description**
AFL [37]	The latest original AFL, version 2.52b
FairFuzz [18]	AFL core with modified path finding strategies
AFL-Byte	Our own implementation by adding more flip-byte operation, in order to find simple path faster
SE-AFL	AFL augmented with Symbolic Execution Techniques

3.6 Evaluation

In order to provide more comprehensive reliable evaluation for fuzzers, we create a testcase bundle to collect the running result of metrics we list above. For the fuzzer tools available with source code, we select AFL and FairFuzz for evaluation. In addition, we also compare the result of our augmented fuzzing with the help of Symbolic Execution as well as AFL-Byte, which is AFL with more byte flip strategy mentioned before in the chapter. Table. 3.4 shows our test objects in this evaluation work.

To test these tools, we design three simple programs with different program features as fuzzing target. The core bug code snippets for the three program are shown in Fig. 3.11.

(a) Bug code in cb1 (b) Bug code in cb2 (c) Bug code in cb3

Figure 3.11: Core bug code snippets for 3 testcases, they all contain divide-by-zero vulnerability and have different settings on path layout and condition constraints

We plan to emphasize the importance roles played by the "Path Layout" as well as the "Difficulty Level of Condition Constraint". We believe different "Path Layout" could cause in different path finding results, which is decided by various path finding strategies, and "Difficulty Level of Condition Constraint" shows the ability of a tool solving constraints for continuing path find.

All three testcases are compiled with GCC 4.8.4 with compile option "-g -O0" as 64-bit ELF programs.

We run all the experiments on one machine with Intel Xeon CPU E5-2697 v3 and 192 GB memory running 64-bit ubuntu 16.04.3 TLS. Each running case use exact one core except SE-AFL require more resources for scheduling and Symbolic Execution work. We repeat each testcase 5 times and pick the median case with respect to finding crash time and path exploration time to reduce the random factor brought by AFL's non-deterministic process.

Table 3.5: Testcases for experiments.

Testcase	**Path Style**	**Condition Constraint**
cb1	Style 1 (A series of true branch)	Easy (1-byte-char)
cb2	Style 1 (A series of true branch)	Harder (2-bytes-char)
cb3	Style 2 (Combination of true&false branch)	Easy (1-byte-char)

The seed input we use for all cases is "asdfgh", which barely has any bias to any tools or target programs.

3.6.1 Running Capability Evaluation

For all key status information, AFL collects and records them into file "plot_data" after every time slot.

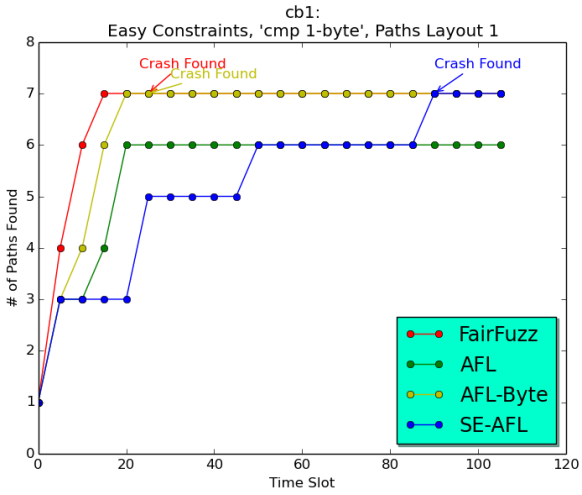
All four tools follow the routines of find paths first and then trigger the bug. Testcase cb1 and cb3 has 7 paths while cb2 has only 4 paths.

Among all the metrics we listed, we found the change of "number of path found over time" quite interesting. By plotting the cumulative number of paths found over time, we can get more comprehensive understanding on each tools. In this experiment, each time slot is 5 seconds.

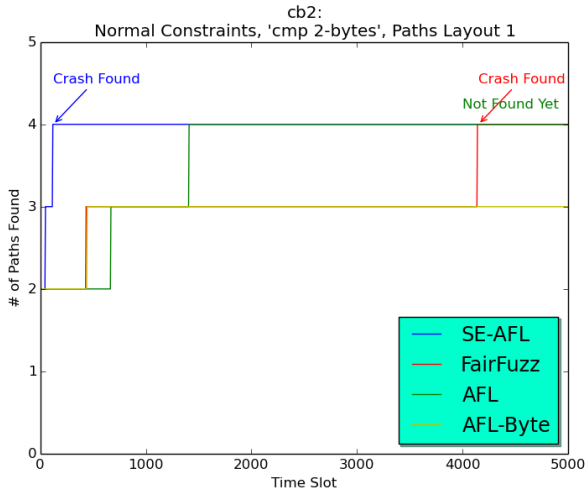
The goal of comparison in Fig. 3.12 is to decide the role "condition constraints" played in fuzzing.

According to Fig. 3.12b, SE-AFL outperforms others (the blue line) when the path constraints changed from "one-byte char" to "two-byte chars". When it comes to complex conditions, extra help such as program analysis or Symbolic Execution is helpful for AFL.

Another noticing finding here is, AFL-byte did the worst in experiment cb2. The one-byte-flip strategy is the reason on failing on two-byte constraints.

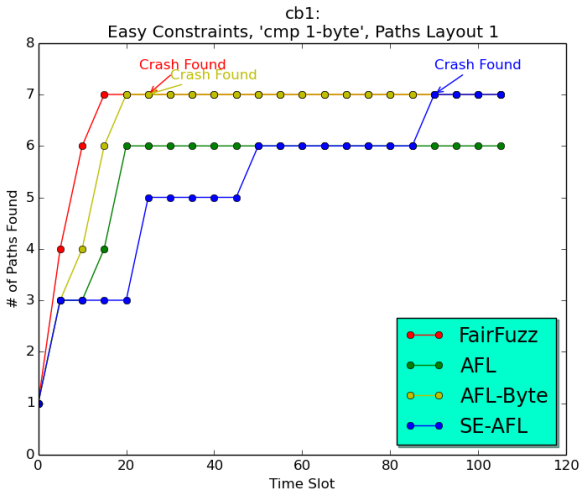


(a) Cumulative number of paths in cb1 found by tools over time

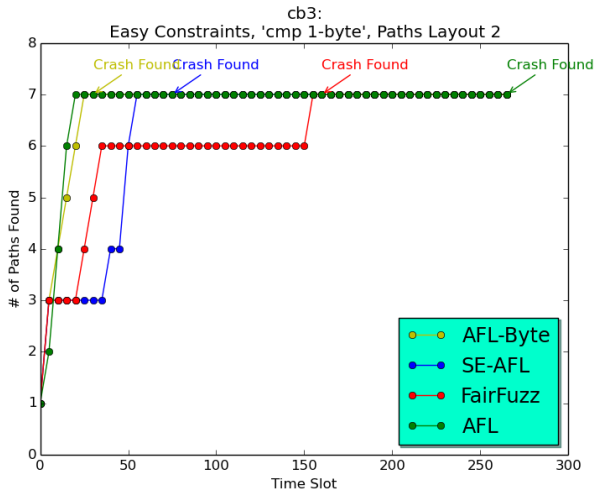


(b) Cumulative number of paths in cb2 found by tools over time

Figure 3.12: Comparison the results of cb1 and cb2, the difference is introduced by the "difficulty level of constraints"



(a) Cumulative number of paths in cb1 found by tools over time



(b) Cumulative number of paths in cb3 found by tools over time

Figure 3.13: Comparison the results of cb1 and cb3, the difference is introduced by the program' "path layout style"

Table 3.6: Using Scope for Analysed Tools.

Platforms&Modes	Linux	x86/x86-64	Greybox
Testable Target	File Format, Library...		
Tiggable CWEs	CWE120,CWE134,CWE369,CWE835...		

The goal of comparison in Fig. 3.12 is to decide the role "path layout style" played in fuzzing.

FairFuzz's modification on path searching strategy is not always working well(the red line in Fig. 3.13b). Each strategy or decision made in path exploration process trades off some cases.

Our AFL-Byte works well on both two cases (yellow lines) because both two cases' constraints are one-byte-char. Comparing to original AFL, it guarantee the finding of new path in deterministic process. When AFL failed to solve one-byte-char constraints in deterministic process, it leaves much more uncertainty to non-deterministic process on finding path quickly. This conclusion shows the potential improvements for AFL's deterministic process in order to get path hit faster, especially for programs with small scales.

SE-AFL is not doing well (blue lines) because this two cases requires rare help from other techniques, and the scheduling work between two techniques causes some delay in finding new paths.

3.6.2 Using Scope Evaluation

The four tools we analysed all have the same AFL core, which decide the using scope as Table. 3.6:

```
// Divide-by-Zero Vulnerability with simpler constraint
char in;
a = 10/(in - 'c');

// Divide-by-Zero Vulnerability with harder constraint
int in;
b = 10/(in - 0xdeadbeef);
```

Figure 3.14: Comparison of vulnerability constraint difficulty level for CWE-365: Divide-by-Zero.

Apart from the common result, SE-AFL has better bug-trigger ability because it can trigger bug code with harder constraints. Fig. 3.14 shows an example of the differences in triggering bug to crash caused by constraint difficulty level.

3.7 Conclusion

In this chapter, we discuss the missing information problem in most of the past fuzzing evaluation works. We first present SRCTF framework to emphasize "Stepwise" and "Reusable" when automated generating binary challenges. We then discuss the possibility of using CTF challenges as fuzzing tools evaluation. We collect the list of necessary metrics as well as a series of programs as testing benchmark. We propose a new benchmake system with testcases focusing on different aspects of program analysis. Our experiment work shows the promising results to make fuzzing tools evaluation more accurate and reliable. Only by understanding and evaluating tools systemically, can we get better improvements on program analyzing and vulnerability finding.

Chapter 4

DEAFL: AFL's Blindspot and Resisting AFL Fuzzing for Arbitrary ELF Binaries

4.1 Introduction

As coverage-based greybox fuzzer, AFL has claimed many successes on fuzzing a wide range of applications. In the past few years, researchers have continuously generated new improvements to enhance AFL's ability to find bugs. However, less attentions were given on how to hide bugs from AFL.

AFL tracks code coverage through instrumentations and it uses coverage information to guide input mutations. Instead of fully recording the complete execution paths, AFL uses a compact hash bitmap to store code coverage. This compact bitmap brings high execution speed but also a constraint: new path can be masked by previous paths in the compact bitmap due to hash conflicts. The inaccuracy and incompleteness in coverage information sometimes prevents an AFL fuzzer from discovering potential paths that lead to new crashes. In general, a potential "interesting" testcase that AFL just mutated may not be synchronized into AFL's queue directory because of the hash conflicts in coverage bitmap.

Research Goals: In this chapter, we aim to answer the following research questions:

- How to explain the situation when an "interesting" testcase cannot be synchronized be AFL?
- What can we achieve by taking advantages of AFL's bitmap hash conflict?

To answer these questions, we first propose “AFL-hash”, an analysis tool to explain if a testcase causes any bitmap hash conflict to a AFL fuzzing instance. With the base of AFL-hash, we introduce Deafl, which can defeat AFL fuzzing instance by creating bitmap hash conflicts and injecting into program. Deafl transforms and rewrites EFL binaries for the purpose of resisting AFL fuzzing. Without changing the functionality of a given ELF binary, the DeafL tool rewrites the input binary to a new EFL executable, so that an easy to find bug by AFL in the original binary becomes difficult to find in the rewritten binary.

4.2 Backgrounds

4.2.1 The Success of American Fuzzy Lop

AFL is successful because of its light-weight instrumentation, effectively generating new inputs and good branch coverage feedback. We know that AFL can act as greybox fuzzers by leveraging light-weight analysis tools on binary code to get the knowledge of target program. In AFL QEMU-mode, with the help of QEMU user emulation mode, AFL can collect enough basic block information to decide if a newly generated input is ”interesting” or not.

In order to explore different paths of the target program, AFL examines newly generated inputs based on branch coverage information, which is known as ”bitmap”. It is a hash table AFL keeps to record the times executed for each branch. For each new execution’s bitmap, AFL compares the specific path bitmap with its global bitmap which contains all branches covered before and will identify such input ”interesting” if the comparison shows differences.

AFL also applies generic algorithms to generate new testcases. Specifically, the mutation strategies are:

1. Deterministic strategies
    (a) Bit flips with varying lengths and stepovers, including byte flips.
    (b) Addition and subtraction of small integers.
    (c) Insertion of known interesting integers(0, 1, INT_MAX, .etc).

2. Non-deterministic strategies:

   (a) Havoc: Stacked bit flips, insertions, deletions, arithmetics

   (b) Splice: Splicing two distinct input files at a random location

In summary, the three keys of AFL's success are:

- Light-weight instrumentation
- Effectively generating new inputs
- Branch Coverage Feedback: Bitmap

With the help of AFL, bugs has been discovered in various kind of programs and software such as Bind, PuTTY, tcpdump, ffmpeg, GnuTLS, libtiff, libpng and so on. Such success makes AFL widely used by most of the Finalist Teams of DARPA CGC 2016.

4.2.2 Traditional Ways to Influence AFL

- Blocking Path Exploration by Adding Complex Constraints Such constraints is usually achieved in the format of "magic bytes".
- Reducing AFL's Running Speed One key of afl's success is running more testcases with less time, reduce program's running speed would directly affect the running speed of AFL.

4.3 Motivation

4.3.1 Problem: Synchronizing Issue when Fuzzing Readelf

During our fuzzing analysis on readelf with augmented AFL, we found an interesting fact: AWhen we combine symbolic execution with AFL, we found AFL refuses to sync several inputs generated by our symbolic execution engine. To reason out this problem, we modified QEMU usermode to collect all basic block transition information like AFL does. For this

```
yinyue@lab301:~/bug_review/coverage$ python afl_hash.py FILE ../readelf_cimfuzz/readelf_x86_64/output
_afl/2069b8c8/seedbox/queue/id\:000178\,70ea5c67
> Step1: is there any new edge?
Yes! 2 new edges found.
('00000000004189dc', '00000000004189e1')
('00000000004189e1', '0000000000417b66')

> Step2: is the bitmap value causing conflicts?
('00000000004189dc', '00000000004189e1')
...... 0x614b1bd0
No conflict
--------------------------------------------------------------------------------------------------
('00000000004189e1', '0000000000417b66')
...... 0x61b98d79
No conflict
--------------------------------------------------------------------------------------------------
Looks good, continue...

> Step3: is the actual bitmap causing conflicts? [MAP_SIZE: 2**16]
('00000000004189dc', '00000000004189e1')
...... Confilct found in location 0x9bd0: existing edges:[('000000000041c9b0', '000000000041c9d1')]
--------------------------------------------------------------------------------------------------
('00000000004189e1', '0000000000417b66')
...... Confilct found in location 0xd79: existing edges:[('0000000000419509', '000000000041951c')]
--------------------------------------------------------------------------------------------------
All the new edges caused conflicts, please try changing the `MAP_SIZE` in afl
```

Figure 4.1: AFL-Hash is able to reveal details of "hash conflict" problem in AFL fuzzing

specific case, we collect all basic block transitions, also known as "edge" in AFL's queue directory and map all edges into AFL bitmap hash table. When our newly generated testcase processed by AFL, we find two new edges which previous queue testcases never touched. After calculating these two edges' hash value in respect of AFL bitmap, we found both two hash values have been taken by other existing edges. This situation is known as "AFL Bitmap Hash Conflict" and it cannot be avoided if the target program is too large.

4.3.2 A New Dimension of Complexities for CTF Challenges

The findings give us an idea: Can we take advantages of such conflict and design harder testcases? Or in other words, can CTF challenges be more resistant to analysis tools such as fuzzers? In the rest of this chapter, we will present our Deafl framework, which can modify

close-source program to resist Qemu mode AFL fuzzing, by introducing hash conflicts to important paths.

4.4 Deafl Design

By given a close-source program, Deafl is aiming at generating codes to cause conflicts with its key path and attach the codes at the end of the program's '.text' section.

4.4.1 AFL-Hash

With the readelf problem we found, we design AFL-Hash to detect bitmap conflicts. An explanation of the readelf conflict problem given by AFL-Hash is showing in Fig. 4.1 AFL-Hash will take the "queue" directory from a fuzzing instance and one or more testcases as input. The goal of AFL-Hash is to decide if the target testcases will bring any bitmap hash conflict with the "queue" directory. It works as following:

- In step 1, AFL-Hash finds out all new edges introduced by the target testcase.
  If there is no new edges found, the target testcase is not "interesting" to AFL. And we will not keep this testcase anymore.
- In step 2, AFL-Hash tries to decide if the related hash value of new edges causing any conflicts.
- In step 3, in respect to the actual AFL bitmap hash value, which is the remainder of calculated hash value devided by bitmap size, AFL-Hash will examine again to decide if there is any hash conflicts.

In Deafl, we extend the function of AFL-Hash to generate codes that contains new edges to cover target edges.

In the example shown in Fig. 4.1, we can see the target testcase(id:000178) has found 2 new edges comparing to AFL's knowledge in "queue" directory. However, both the 2 edges

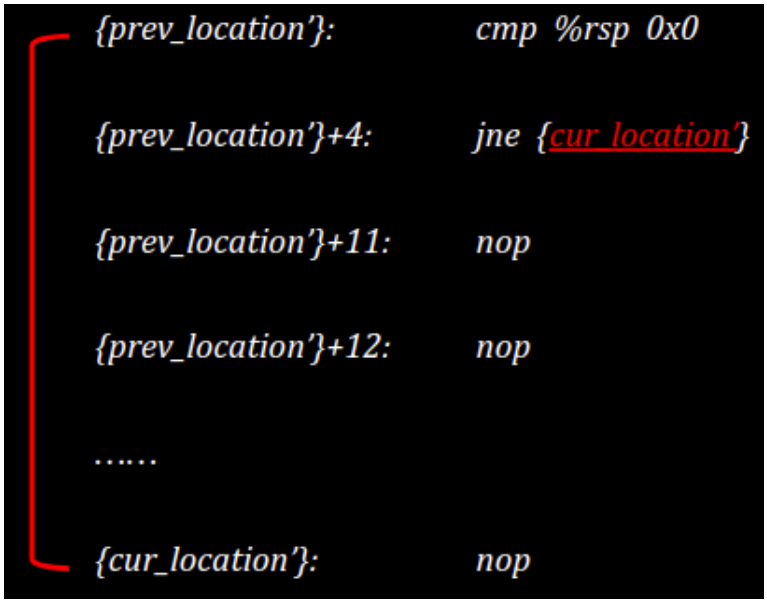


Figure 4.2: A fake edge example created by afl-hash

has found conflicts in step 3. As a result, the testcase(id:000178) cannot be synchronized into AFL's "queue" directory even though it actually introduce new behaviours to AFL.

Inspired by this finding, given target hash value starting address for injecting, AFL-hash can find out a relative short address tuple (prev_location‘, cur_location‘) as Fig. 4.2.

4.4.2 Add-Text

Once the code has generated, we need to inject them into a compiled binary file, the modified program should also be runnable and not affected by the new code we injected. To defeat QEMU-mode AFL, the codes need to be injected into the “.text” section. Our design is to inject code at the end of ".text" section and modify the program's entrypoint to the beginning of our code. At the end of our code, it will jump back to the program's original entrypoint and execute the original program flows.

The challenges we are facing here is binary rewriting/patching without source code. Binary reassembling technique has been discussed in [35, 34]. Our "add-text" works as following:

- Locate the end of ".text" section and insert our code generated by AFL-Hash.
- Calculate the offset and padding size introduced by our code and move all sections after ".text" section.
- Update all pointer/address reference in necessary sections with the offset adjusted.

  Such necessary sections including: ".dynamic", ".rela.dyn", ".rela.plt", ".symtab", ".dynsym"

  For aligned data in section:".text", ".data", ".rodata", if it falling in the program address we modified, we will update it with calculated offset.
- Update ELF table information, including Segment Table, Section Table as well as the new program entrypoint, new file offset for different table.

4.4.3 Implementation

The Deafl framework is the combination of afl-hash and add-text. We write a python script to realize these functionalities, with the help of:

- Library LIEF, to analyse program's ELF data
- Modified user-mode QEMU, to collect path coverage information as AFL does.

We modify target program by injecting our own code containing fake edges to cause bitmap hash conflicts. The new program will be executed as Fig. 4.3 shows.

Based on the new entrypoint we modified, the program will first execute our injected code. At the end of our code, we will jump back to the original entrypoint for original program control flow. AFL's path coverage information will keep updated for these two

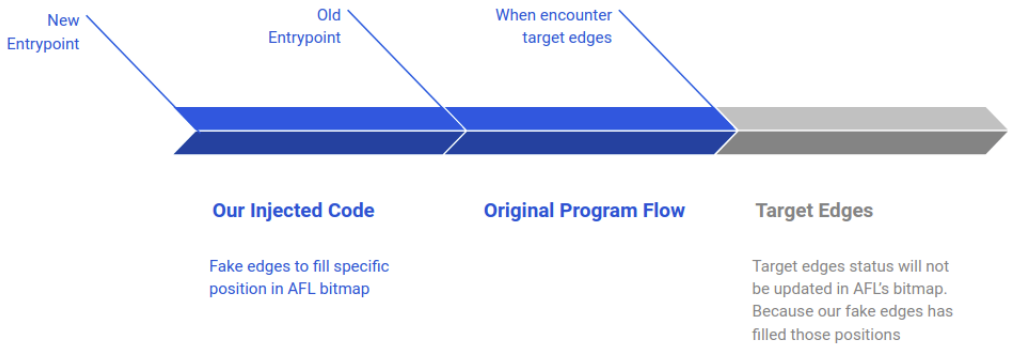


Figure 4.3: The executing flow after modifying program

parts execution. When encounter target edges, AFL's bitmap will not be updated, because such bitmap index calculated by the edges have been taken by one of the fake edges we injected.

When given target edges list, Deafl can start modifying target binary as Fig. 4.4 shows. In practical, such edges list can be achieved by performing CFG analysis on program. In practice, Deafl can be use to block specific testcase which is closer to crash input.

4.5 Case Studies

We now show some results of applying Deafl to CTF challenges or vulnerability programs. All of the cases is tested with latest version of AFL v.2.52b. Our goal in this case study is to evaluate the influence Deafl introduced to AFL's fuzzing. We picked 3 different 64-bit programs for the case studies. The Deafl workflow in analysing and modifying target binary is shown in Fig. 4.5.

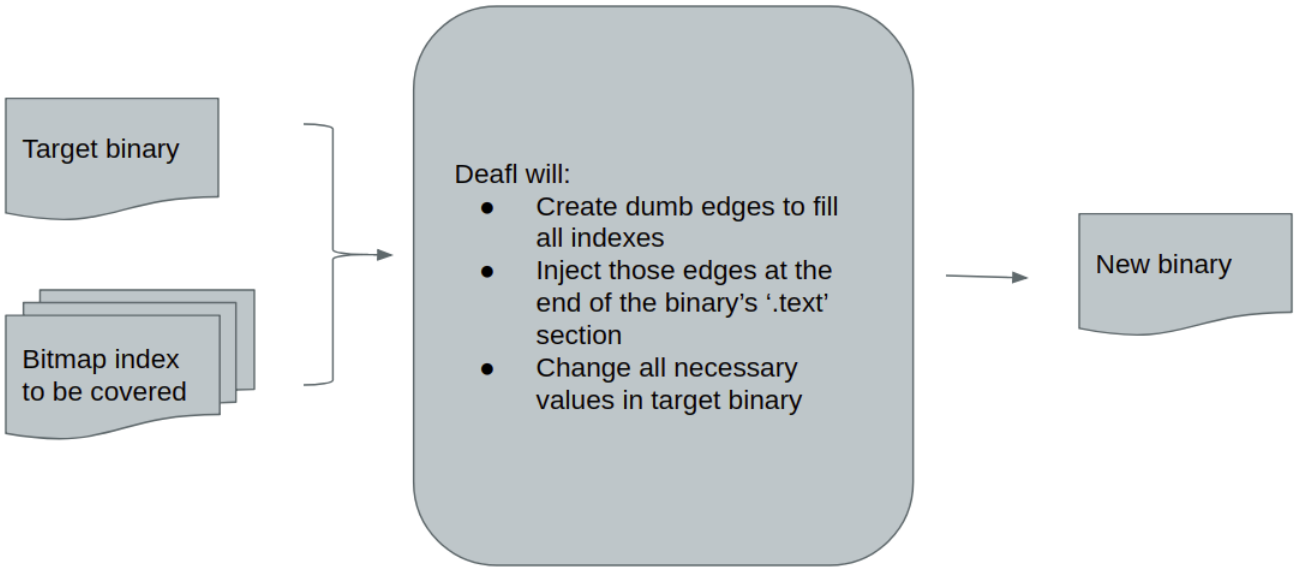


Figure 4.4: The workflow for Deafl when target edges provided

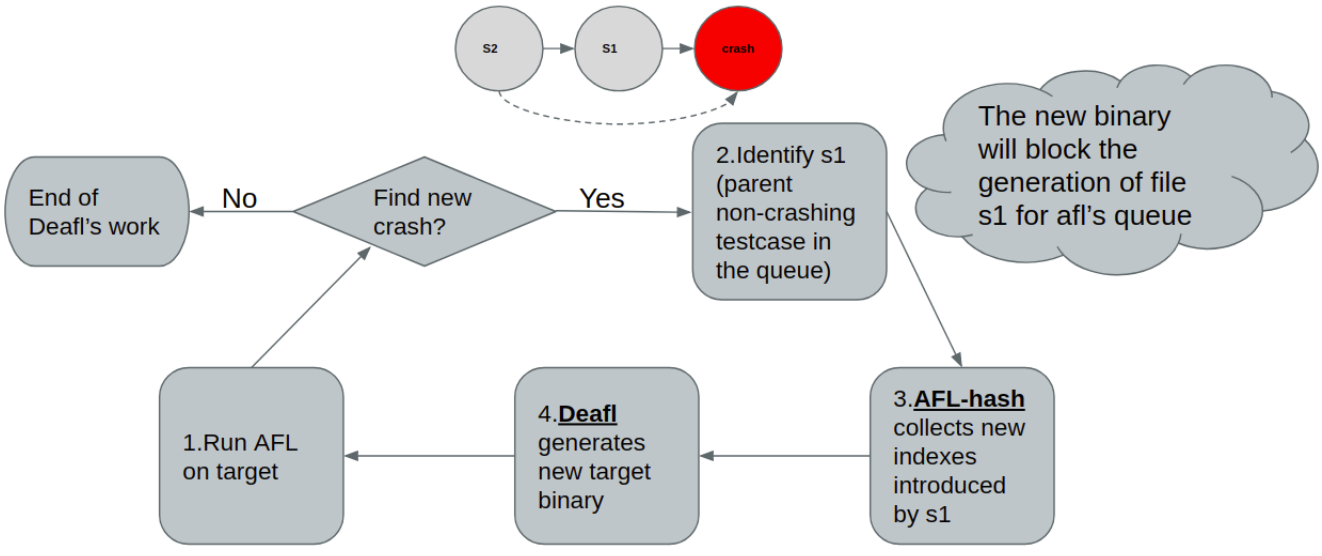


Figure 4.5: The workflow for Deafl case studies

Table 4.1: Case 1 Comparison.

file	**size**	**time to crash(min)**	**paths found when crash**	**exec per sec**
(old) cb	12K	4	7	3041.45
(new) cb_c246be	784K	3000	6	2686.27

4.5.1 Example 1: MAGIC!

In the first case, we show the ability of Deafl by presenting a simple program "MAGIC!". As Fig. 4.6 shows, the core of this program has a divide-by-zero bug, which can be triggered if the first six bytes of input is "MAGIC!"

The input seed we provided is "asdfgh", which in this case study, will not cause any bias to the evaluation. It takes AFL less than 4 minutes to find the crash. The way AFL finds the crash is mutating testcase "MAGIC*" to "MAGIC!", which is a "one-byte" searching space.

Now we applying Deafl to this program to block the generation of "MAGIC*". Deafl will first analyse the new basic block transitions introduced from "MAGI**" to "MAGIC*", and then it creates own edges that have same bitmap hash values with those transitions. In this way, AFL will get bitmap conflicts when meeting testcase like "MAGIC*", thus it will no longer take "MAGIC*" into "queue" directory because it is not "interesting" at all. The modified "MAGIC!" fuzzing takes more than 50 hours to get the bug.

To explain this case, we compare the "queue" directory of this two runs which is shown in Fig 4.7. After Deafled, the queue directory will not take input "MAGIC*". The distance from testcase id:5("MAGI**") to crash input("MAGIC!") is 2 bytes, making the chance of mutation:

$$1/2^{16} = 1/65536$$

```
int DBZ(uchar* out){
    int a = 0;
    int ret = 0;

    if(out[0] == 'M'){
        if(out[1] == 'A'){
            if(out[2] == 'G'){
                if(out[3] == 'I'){
                    if(out[4] == 'C'){
                        a = 10/(out[5]-'!');
                        ret = 5;
                    }
                    else{
                        ret = 4;
                    }
                }
                else{
                    ret = 3;
                }
            }
            else{
                ret = 2;
            }
        }
        else{
            ret = 1;
        }
    }
    else{
        ret = 0;
    }

    return ret;
}
```

Figure 4.6: Vulnerability code snippet for program "MAGIC!"

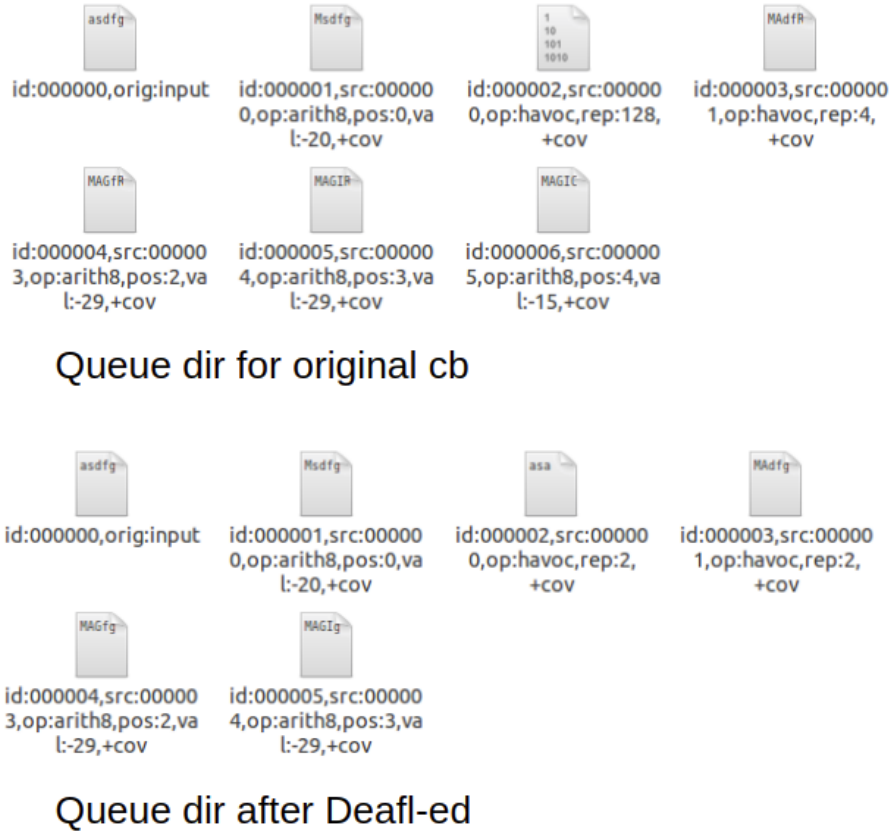


Figure 4.7: Comparision for two "queue" directory

is smaller, comparing to the 1 byte case:

$$1/2^8 = 1/256.$$

4.5.2 Example 2: CVE-2015-3138, tcpdump

To ensure Deafl's functionality, we take tcpdump as target in this case. The vulnerability we use is CVE-2015-3138 [23]. It is a vulnerability of out-of-bound pointer access because of input validation failing and it can be used to cause denial-of-service attack. We pick tcpdump version 4.7.3 as our target program because the CVE affects all tcpdump before 4.7.4. We

Table 4.2: Case 2 Comparison.

file	**size**	**time to crash(min)**	**paths found when crash**	**exec per sec**
(old) tcpdump	2.0M	5	298	565.85
(new) tcpdump _b531c4	2.9M	1169	3049	533.86

carefully picked an input as the seed testcase, so that AFL can find the crash and find it fasterer. It takes less than 5 minutes to get the crash.

We now evaluate how Deafl works in this case. According to the first run result, we identify identify the source input of the crash testcase, which is "id:22" in queue directory. We apply Deafl on tcpdump program by blocking the generation of "id:22". The Deafled tcpdump will not synchronize testcase like "id:22" into "queue" directory because all new edges introduced by "id:22" has been blocked through bitmap conflicts.

According to result shown in Table. 4.2, we can see now it takes more than 19 hours to finally mutate the crash testcase. To dig deeper the influence Deafl introduces, we list the two "queue" directories as Fig. 4.8

We know that at the beginning of AFL's running, AFL-fuzz will apply deterministic strategies such as bit-flip or byte-flip to mutate seed input. Such strategies is fixed once the seed input is given. In this case, every time we run afl, it will mutate the seed input by flip each position of the file and check if there is new "interesting" input. The flip strategies on position 56 will always get a new "interesting" input, as id:22 in Fig. 4.8[a].

We pick id22 as Deafl's blocking list because it can mutate the crash input. After we appled Deafl on tcpdump, flip strategies on position 56 cannot get new "interesting" testcases because such synchronization is blocked by afl. This blocking directly delay the mutating of crash input. AFL need to explore far more paths to get closer to the crash point.

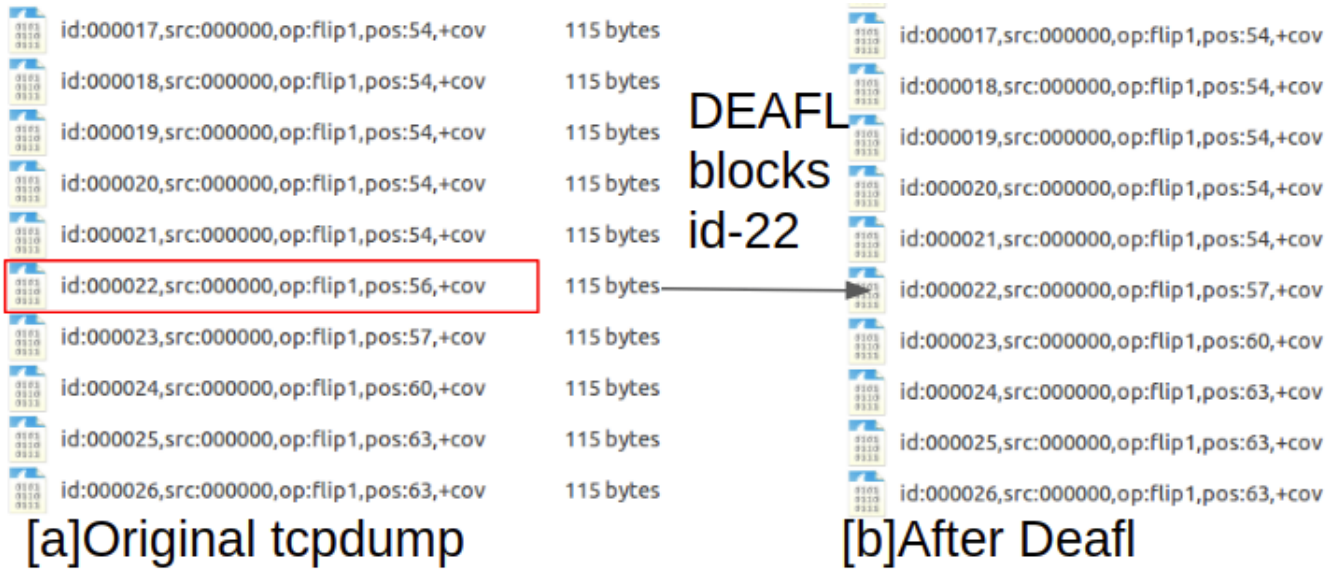


Figure 4.8: Deafl blocks the synchronizing of old "id22"

4.5.3 Example 3: CVE-2018-10534, objcopy

Last case we picked is a recent CVE found by our group, CVE-2018-10534 [25]. This is a out-of-bound write bug and it was found when fuzzing objcopy of binutils using augmented AFL.

We take one of the closest queue input as the seed input and running AFL fuzzing on original objcopy. In this way, afl can find a crash within 1 minute. Now we block the synchronization link between seed input and the source input to crash using Deafl, which can postpone the finding of crash. The result is shown in Table. 4.3.

4.6 Discussion

4.6.1 Limitation

We use 3 cases to demonstrate Deafl's ability to block the synchronization of specific testcase, so that the crash input mutated from such testcase is harder to be found.

Table 4.3: Case 3 Comparison.

file	**size**	**time to crash(min)**	**paths found when crash**	**exec per sec**
(old) objcopy	4.3M	1	164	539.6
(new) objcopy _9b64fd	5.2M	567	3019	503.9

In general, the deafL tool needs to provide answers to these 3 questions

- Which edges to target (to create hash conflicts)?
- How to create an edge that has a specific hash value?
- How to inject fake edges to a binary?

Ideally, we can add fake edges to completely fill the whole AFL's shared memory. Such method will turn the new binary file quite large and running slow. The practical goal is to find those edges that lead to the mutation of crash inputs, which can be achieved by analyzing AFL's running result on target program. All edges that link between the initial seed inputs and the targeted seed files will be Deafl's target edges to block.

The more target edges we found, the more code we have to inject into the program. As Table. 4.4 shown, we list the file size overhead and the crash time increment introduced by Deafl.

In these three cases, the new .text data Deafl injected is less than 1M. Such file size overhead could be increased if the number of target blocking testcases or the number of blocking edges gets larger. Those file overhead will reduce the execution speed of fuzzing tools. In extreme cases, deafl may have to block a large amount of edges if there is a large path space after program's crash point. Which makes Deafl less effective. Such scalability

Table 4.4: Comparing file size overhead and crash time increment introduced by Deafl

Case	**old file size**	**new file size**	**file overhead**	**old crash time(min)**	**new crash time(min)**	**time increase**
1 MAGIC!	12K	784K	6400%	4	3000	74900%
2 tcpdump	2.0M	2.9M	45%	5	1169	23280%
3 objcopy	4.3M	5.2M	20.93%	1	567	56600%

Table 4.5: Comparing execution speed reducing introduced by Deafl

Case	**old exec per sec**	**new exec per sec**	**speed reduced**
1 MAGIC!	3041.45	2686.27	11.67%
2 tcpdump	565.85	533.86	5.65%
3 objcopy	539.6	503.9	6.61%

problem is also faced by AFL itself, larger target program will bring more conflict tuples and running overhead when running under fuzzer.

The injection of code will also cause the reduction running speed, as Table. 4.5 shows.

Another problem for current Deafl framework is that our injected code has specific features, which can be detected by experienced user. Program rewriting as well as obfuscation could resolve the issue.

4.6.2 Future Work

We've demonstrate the usage of Deafl by providing testcases to be blocked. Deafl also works by directly feeding list of edges. This requires extra analysis work to collect all the necessary edges as block target.

On the other hand, the algorithm to generate injected code can be improved so as to reduce the overload introduced by Deafl. In ideal case, we hope to find a way to complete poison AFL's bitmap. In that way, all bitmap index has been taken and it will totally break AFL's path coverage knowledge. Every testcase processed by AFL will be regarded as "uninteresting" thus no new testcases will be stored into AFL's queue directory.

We propose the idea of defeat AFL by exploiting bitmap hash conflict and make it work with our design. However, the way we generate and inject code can be detected by experienced end user with the help of other analysis techniques. A better method to hide the trace of injection is also considered as one of the important future work.

The current Deafl framework only works in resisting AFL-Qemu mode. Other instrumentation schema such as Intel-PT, PIN, DynamoRIO requires extra future work on finding the design flaw in path coverage tracking algorithm.

4.7 Conclusion

In this chapter, we discuss one of the limitations faced by AFL. AFL's high efficiency comes from its compact data structure for edge coverage. However, bitmap hash conflict creates a blindspot for AFL. This chapter demonstrates such limitation with examples showing how the blindspot limits AFL's ability in finding bugs, and how it prevents AFL from taking seeds generated from complementary approaches such as symbolic execution. We present Deafl, which Intentionally create hash conflicts for edges that lead to the mutation of crash inputs, to add fuzzing resistance to ELF programs. We believe this technique can be used as a feature of AFL resistance for future CTF challenges.

Chapter 5

Designing CTF Challenges with a New Dimension of Complexities: Features to Resistance Program Analysis Tools

5.1 Introduction

Capture the Flag(CTF) competitions have become popular today. Good CTF exercises are not only good approaches for learning cyber security concepts and technical in fun, but also good tools to measure all kinds of popular vulnerability discovering, exploiting, patching technologies. According to popular CTF website CTFtime, usually there were more than one hundred CTF events a year(141 in 2017, 109 in 2016). Participating in multiple events can bring good practice for both students and security experts. However, with the growing trend of automated vulnerability detection techniques used to solve CTF challenges, the need of more and better CTF challenges is urgent. Such challenges should not only keep all typical CTF characteristics but also require more state-of-the-art program analysis tools to solve. The large demand for more and good CTF challenges poses a new challenge to CTF designers because most of the CTF generation work are expensive.

Research Goals: In this chapter, we aim to answer the following research questions:

- What should we emphasize when the CTF challenge consumers are program analysis tools such as GDB?

To answer the question and make our CTF challenges valuable to program analysis tools, we introduce a new dimension of complexities for CTF challenges: Program Analysis Tools Resilience, which aims to tune the CTF difficulty level of applying various analysis tools. We have discussed modifying CTF challenges to emphasize AFL's fuzzing blindspot in chapter

4. In this chapter, we would introduce CTF challenges design aiming at block the use of program analysis tools, GDB specifically, by modifying ELF metadata. When fuzzing program analyzing tools(gdb, objcopy, readelf...), we realized that some ELF header metadata, which are usually consumed by these tools, will not affect the running of the program, but can affect the functionality during program analyzing.

5.2 Related Work

5.2.1 Debugger

A debugger is a tool to give user a view of the running program in a natural and understandable way, and provide control over the execution as well as useful information. With the help of debuggers, we can interact with and examine the state of running program by setting breakpoints and examining memory and register contents. It requires the debugger coordinating with the compiler, which converts human-readable source code into machine language, to make a program debuggable. The GNU Debugger (gdb) is the standard debugger for most Linux systems.

5.2.2 The DWARF format

DWARF is a complex format[8], building on many years of experience with previous formats for various architectures and operating systems. It has to be complex, since it solves a very tricky problem - presenting debugging information from any high-level language to debuggers, providing support for arbitrary platforms and ABIs.

Take a C program as an example, when the source code compiled with "-g" option, GCC will generate debug information in DWARF format and store it into the executable program. By dumping the ELF's section information using "readelf" or "objdump", we can find several sections with names starting as ".debug_". These are the DWARF debug sections generated by GCC:

An example of readelf output for ".debug_" sections

```
[28] .debug_aranges PROGBITS 0000000000000000 0020ee 000030 00 0 0 1
[29] .debug_info PROGBITS 0000000000000000 00211e 0003e2 00 0 0 1
[30] .debug_abbrev PROGBITS 0000000000000000 002500 000158 00 0 0 1
[31] .debug_line PROGBITS 0000000000000000 002658 0000ea 00 0 0 1
[32] .debug_str PROGBITS 0000000000000000 002742 0002c8 01 MS 0 0 1
```

And the debugger can use these sections to provide detailed information for debugging and analyzing. DWARF uses a series of debugging information entries (DIEs) to define a low-level representation of a source program. Each debugging information entry consists of an identifying tag and a series of attributes. An entry, or group of entries together, provide a description of a corresponding entity in the source program. The tag specifies the class to which an entry belongs and the attributes define the specific characteristics of the entry.

In addition, there is another ELF section containing DWARF data, which is called ".eh_frame". This section contains information necessary to implement frame unwinding. For each instruction in program, this section can be used to specify how to compute the location on the stack of the return address. Stack unwinding is useful during debugging, program analyzing, and also for the C++ runtime exception handling. The format of the .eh_frame section is similar in format and purpose to the .debug_frame section, which is specified in DWARF debugging information format. The fact is, the format of the data in the ELF .eh_frame section is based on DWARF's Call Frame Information format, with additional information in the CFI augmentation fields, and this part is actually defined in C++ ABI implementation. The dwarf data in this section are more sensitive and unlike the ".debug_" sections, it cannot be remove using "strip". This section creates another space for potential DWARF format misusing.

An example of readelf output for ".eh_frame"

```
00000090 0000000000000044 00000064 FDE cie=00000030
pc=0000000000400580..00000000004005e5
LOC CFA rbx rbp r12 r13 r14 r15 ra
0000000000400580 rsp+8 u u u u u u c-8
0000000000400582 rsp+16 u u u u u c-16 c-8
0000000000400587 rsp+24 u u u u c-24 c-16 c-8
000000000040058c rsp+32 u u u c-32 c-24 c-16 c-8
0000000000400591 rsp+40 u u c-40 c-32 c-24 c-16 c-8
0000000000400599 rsp+48 u c-48 c-40 c-32 c-24 c-16 c-8
00000000004005a1 rsp+56 c-56 c-48 c-40 c-32 c-24 c-16 c-8
00000000004005ae rsp+64 c-56 c-48 c-40 c-32 c-24 c-16 c-8
00000000004005da rsp+56 c-56 c-48 c-40 c-32 c-24 c-16 c-8
00000000004005db rsp+48 c-56 c-48 c-40 c-32 c-24 c-16 c-8
00000000004005dc rsp+40 c-56 c-48 c-40 c-32 c-24 c-16 c-8
00000000004005de rsp+32 c-56 c-48 c-40 c-32 c-24 c-16 c-8
00000000004005e0 rsp+24 c-56 c-48 c-40 c-32 c-24 c-16 c-8
00000000004005e2 rsp+16 c-56 c-48 c-40 c-32 c-24 c-16 c-8
00000000004005e4 rsp+8 c-56 c-48 c-40 c-32 c-24 c-16 c-8
```

As the example shown above, the encoded data in ".eh_frame" section describe a large table. The rows of the table are the instruction address in program's text, while the columns correspond to different registers. Each line tells the corresponding address how to restore the entry in the previous call frame, which is the process of stack unwinding.

Oakley et al. [27] describe ways in which an attacker can modify a program's exception handling tables to execute arbitrary programs. In brief, a program location in the CFI is described using a DWARF Location Expression which may be a DWARF expression written in DWARF byte code. A malicious attacker who can modify an executable can insert an

arbitrary program into the exception handler tables and perform essentially any operation permitted by the run time byte code interpreter.

5.3 Add Debugger(GDB) Resistance Feature To a Program By Exploiting ELF Metadata

Now facing the new challenge, a good CTF challenge should reveal latest security topics as well as state-of-the-art techniques.

Base on automated CTF generation framework, we can add features for more purpose. By applying Deafl technique introduced in Chapter 4, we can make a CTF challenge resistant to fuzzing tools like afl-fuzz.

When fuzzing program analysing tools(gdb, objcopy, readelf...), we realized that some ELF header metadata, which are usually consumed by these tools, will not affect the running of the program, but can affect the functionalities during program analysing. Our method to influence the use of such tools is make unstripped binary challenge and modify specific metadata.

Utilizing non-".text" sections such as DWARF format metadata, has been proved useful to gain the control of the execution flow for malicious purpose [27]. We suggest such idea can also be used for reference when designing CTF challenges. Malformed metadata may not affect the running of the challenge program at all but can precisely defeat the usage of some program analysis tools.

Traditional challenges usually released in the format of stripped binary, or binary as well as the source code. We present the new format of releasing program, un-stripped challenges. In the rest of this chapter, we will show the features that could affect the use of gdb when solving CTF binary challenges, based on the findings when learning DWARF format as well as fuzzing gdb.

```c
#include <stdio.h>

int main(){

    int a;
    a = 12345678;
    if (a > 79)
        printf("helloworld: %d\n", a);
    else
        printf("bingo: %d\n", a);
    return 0;
}
```

Figure 5.1: A simple C program example

5.3.1 Modifying DWARF format data in ".debug_" sections to resist GDB

If an ELF file has its debug symbols stored in the ".debug_" sections, program analyzing tools such as GDB can take that information providing more help during the debugging process.

Case 1

Scenario: Take the C program shown in 5.1 as an example. Suppose our debugging work with this program is set an breakpoint before the "if" condition(Line 7), and using command "print" to check the value of variable "a".

As the output shown below, GDB print out the information on variable correctly.

Output of debugging with normal debug symbols

```
(gdb) b 7
Breakpoint 1 at 0x40053c: file hello_int_if.c, line 7.
(gdb) r
Starting program: /home/ctf/gdb_demo/bad_dwarf/hello_int_if


Breakpoint 1, main () at hello_int_if.c:7
7 if (a >79)
(gdb) p a
$1 = 12345678
(gdb) p/x a
$2 = 0xbc614e
(gdb) p sizeof(a)
$3 = 4
(gdb) c
Continuing.
helloworld: 12345678
```

However, by only changing one single byte in the ELF file, we can see something interesting.

Output of debugging with modified debug symbols

```
(gdb) b 7
Breakpoint 1 at 0x40053c: file hello_int_if.c, line 7.
(gdb) r
Starting program: /home/ctf/gdb_demo/bad_dwarf/bad_hello_int_if


Breakpoint 1, main () at hello_int_if.c:7
7 if (a >79)
(gdb) p a
$1 = -4431538
(gdb) p/x a
$2 = 0xbc614e
(gdb) p sizeof(a)
$3 = 3
(gdb) c
Continuing.
helloworld: 12345678
```

We can notice that, despite the running result after continuing is still correct, this time, GDB fails to provide correct information about variable "a". It shows a negetive value "-4431538" instead of "12345678", while the hex value of a is still the same as the normal one. The main reason of this issue is gdb treats the size of "int" type variable "a" as "3*8=24" bits, which should be "4*8=32" bits. As the comparison shown in 5.2, the only difference of this two file is one byte with value "0x04" or "0x03".

The modified byte at offset 0x10F3 lies in the range of section ".debug_info" of the ELF file

```
[28] .debug_info PROGBITS 0000000000000000 0000109b
000000000000009e 0000000000000000 0 0 1
```

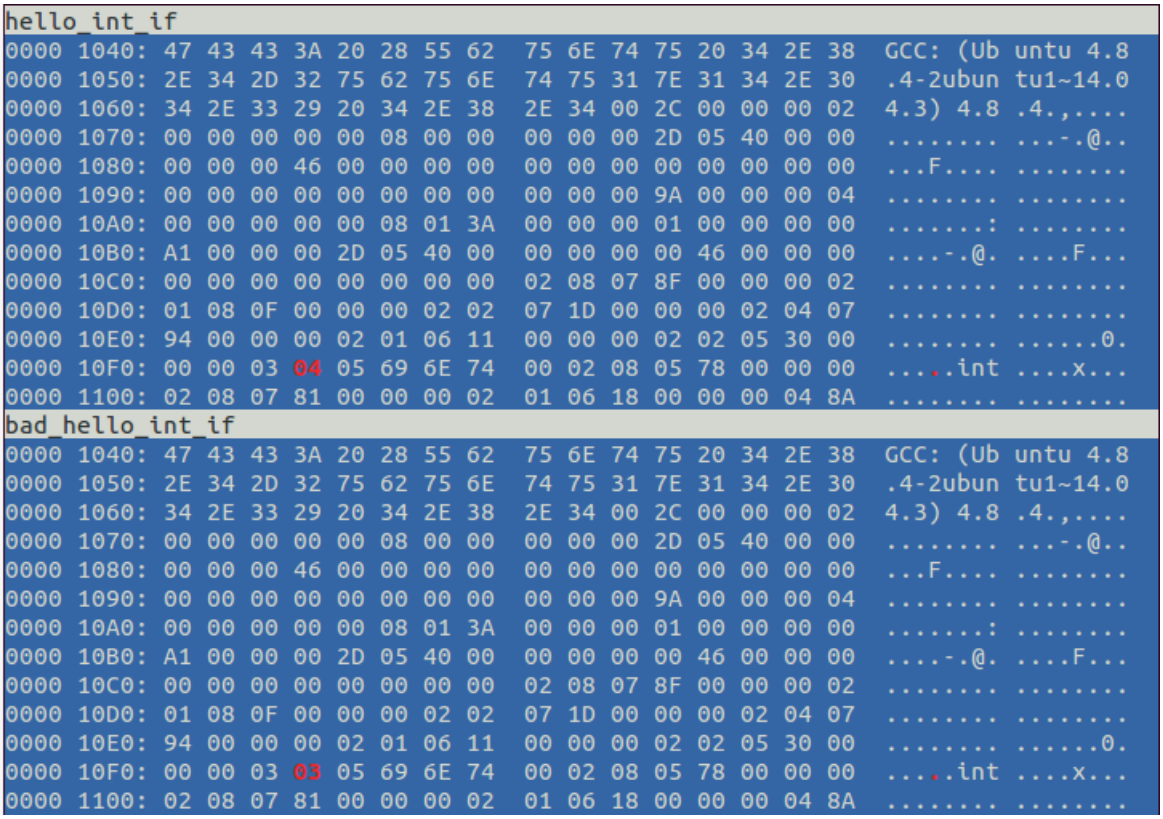


Figure 5.2: Comparison of the two ELF file

The differences reflect in .debug_info variable base type

<1><57>: Abbrev Number: 3 (DW_TAG_base_type)

<58>DW_AT_byte_size : **4**

<59>DW_AT_encoding : 5 (signed)

<5a>DW_AT_name : int

<1><57>: Abbrev Number: 3 (DW_TAG_base_type)

<58>DW_AT_byte_size : **3**

<59>DW_AT_encoding : 5 (signed)

<5a>DW_AT_name : int

Compilers and debuggers are supposed to share a common understanding about whether and int type is 16 or 32 or 64 bits. It is especially useful when one single hardware can support multiple size integers. In order to achieve compatibility, DWARF format defines base types and store such information into the ".debug_info" section. Malformed data of such information will mislead the using of GDB because GDB totally trusts debug information and use it without any checking. The type confusion we found is one of the most common cases of misleading GDB, which make the debugging work harder. Nevertheless, those modified debug information may have no effect on the running of the target program in GDB.

In summary, the information stores in ".debug_" sections can tell program analysis tools information such as: source files(path and name), names (of functions, variables, auguments), base type descriptions, mapping between source file and machine instructions and so on. Tools like GDB will use the information directly without any checking. Although we have not found any harmful consequence on using modified evil DWARF data, the total trust of malformed data will make debugging work harder.

In order to prevent stripping program's data which includes our features, we can try modify the ELF header information to make strip not working. In such case, strip will detected corrupt data and stop its work, leaving the binary unchanged after running strip.

5.3.2 Modifying DWARF format data in .eh_frame section to resist GDB

Besides sharing basic information from compilers to debuggers, another main purpose of using DWARF format data is to show how to unwind stack. This part of work is done by DWARF data stored in ".eh_frame" section. This part of data is much more sensitive and could lead to bad consequences by misusing. The Dartmouth's work [27] has shown a case on exploiting such DWARF by leveraging C++ exception handling process. We would like to try something in the GDB debugging environment.

Case 1: Malformed DWARF data can lead GDB crashing, with a potential risk of exploiting

Inspired by our fuzzing result on gdb, we found the CVE-2017-9778 [24] : GDB 8.0 and earlier fails to detect a negative length field in a DWARF section. A malformed section in an ELF binary or a core file can cause GDB to repeatedly allocate memory until a process limit is reached. This can, for example, impede efforts to analyze malware with GDB.

This vulnerbility happens exactly in file gdb/dwarf2-frame.c in function "decode_frame_entry()". when consuming the malformed DWARF data, GDB failed on checking the payload before consuming it. Our modified DWARF data had a negative size value and it triggered an integer-overflow vulnerability in GDB.

Even though in most cases, malformed DWARF does nothing as harmful as crashing GDB during the consuming process , it can also bring bad influences when guiding GDB for stack unwinding.

Case 2: Malformed DWARF data can lead GDB performing wrong when unwinding stack

Scenario:

We now have a program to be analyzed(it does not matter if the debug information in ".debug_" sections exist or not), our debugging procedure is to set a break point at a function called "sayHello" then execute the program. When GDB hits the breakpoint, we would like to see the current functions call frame by using command "bt"(short for "backtrace").

Output of printing backtrace of normal ELF file

```
Breakpoint 1, sayHello () at demo.cpp:7
7 printf("Hi Everyones");
(gdb) bt
#0 sayHello () at demo.cpp:7
#1 0x0000000000400ac8 in doStuff () at demo.cpp:39
#2 0x0000000000400b3e in main (argc=1, argv=0x7fffffffe2d8) at demo.cpp:60
(gdb)
```

As the output shown above, the call frame for current breakpoint is main() ->doStuff() ->sayHello(). When we perform the same operations on a modified file, we get this output:

Output of printing backtrace of our modified ELF file NO. 1

```
Breakpoint 1, sayHello () at demo.cpp:7
7 printf("Hi Everyone");
(gdb) bt
#0 sayHello () at demo.cpp:7
#1 0x0000000000400ac8 in doStuff () at demo.cpp:39
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb)
```

After locating the specific FDE for target function sayHello() in the ".eh_frame" section, we can modify the data so that the stack unwinding table changed. In this way, GDB will use wrong information as guidance for unwinding work. When calling command "backtrace", the wrong unwinding triggered and GDB detected stack got corrupted. In this way, backtrace information is no longer useful, thus making debugging work much harder.

By modifying the DWARF data properly, we can slightly change the unwinding result without letting GDB know something is wrong.

Output of printing backtrace of our modified ELF file NO. 2

```
Breakpoint 1, sayHello () at demo.cpp:7
7 printf("Hi Everyone");
(gdb) bt
#0 sayHello () at demo.cpp:7
#1 0x00000000006020e0 in ?? ()
#2 0x00007fffffffe1d0 in ?? ()
#3 0x0000000000400b3e in main (argc=1, argv=0x7fffffffe2b8) at demo.cpp:60
(gdb)
```

In the case above, we successfully modified file NO. 2, and the backtrace records from 3 layer to 4 layer, and hide the occurrence of function "doStuff()".

DWARF data for stack unwinding is much more sensitive than the debugging symbols. Combining its bytecode instructions can get arbitrary expressions accessing registers and memory locations. Such operations can do harmful things in scenarios like C++ runtime exception handling. Alghouth such exploitation has not proved working in GDB debugging process, all program analysis tools should be careful before using the untrusted data.

5.3.3 Different scenarios to resist GDB

- Trigger more crashes when using gdb:

  Besides the CVE we mentioned above, more crashes found when fuzzing GDB. Vulnerabilities like this would result in crash when running gdb. In addition, if we can exploit one of vulnerabilities, we can hijack the running flow of GDB, making the analyzing and debugging work impossible.

- Hang when using gdb:

Inspired by hangs we found when fuzzing gdb. Except the hangs cause by time out, we do find a hang status triggered by infinite loop. Challenges with this feature can deny the service of GDB.

- Wrong information when using gdb:

  Instead of deny the service of GDB, proper modification on DWARF information will give GDB wrong information because GDB trusts the information without any checking. Information about variables, structs, classes, functions would be poisoned, thus cannot be used for analyzing or debugging.

With the ELF metadata analysis on GDB process, we stress the potential security risk in unexpected places. The ELF format is complex, analysis tools like GDB should be careful when handling its payload.

5.4 Add Features to Resilient Program Analysis Tools

On the bright side, we can introduce the potential risk we found into CTF challenges to raise the awaress of better security computing as well as more powerful techniques to counter them. This is exactly the objective of CTF: A good CTF challenge should reveal latest security topics as well as state-of-the-art techniques.

Base on automated CTF generation framework introduced in Chapter 3, we can add features for more purpose. By applying Deafl technique introduced in Chapter 4, we can make a CTF challenge resistant to fuzzing tools like afl-fuzz. When fuzzing program analysing tools(gdb, objcopy, readelf...), we realized that some ELF header metadata, which are usually consumed by these tools, will not affect the running of the program, but can affect the functionality during program analyzing. Unstriped program with more metadata such as debug symbols can be a new format of releasing programs or CTF challenges. Sometimes more does not means better.

Both anti-fuzzer and anti-debugger features has the same objective: to prevent or delay the usage of Program Analysis Tools on target program. CTF challenges hardened with such features will make it more difficult for other parties to understand it, which raise higher requirement for better techniques and tools on program analyzing.

5.5 Conclusion

In this chapter, we introduce our thoughts on exploring new dimension of complexities to CTF challenges to provide program analysis tools resistance. We discuss the possibility of misleading or disabling the use of GDB by inputing malformed metadata, especially DWARF format data in ELF challenge files. Combining with Deafl work in Chapter 4, which can defeat AFL by hardening CTF challenges, we propose the new dimension of CTF complexities: "Resistance to Program Analysis Tools". We believe this design can put forward the improvements of better tools and techniques.

Chapter 6

Summary

Capture the Flag(CTF) games and competitions have become popular today. Good CTF exercises are not only good approach for learning Cyber Security concepts and techniques in fun, but also useful benchmarks to measure all kinds of popular vulnerability discovering, exploiting, patching techniques or tools. Harder challenges require more sophisticated techniques or tools. Since automated program analysis techniques such as fuzzing and symbolic execution has been proposed and wildly applied, good ground-truth corpora with vulnerability is required for comprehensive evaluation on those tools. The CTF challenges have been exclusive to human competitors until DARPA sponsored the Cyber Grand Challenges(CGC), a CTF competition to showcase the current automatic program analysis techniques. DARPA's CGC addresses the need for more datasets and evaluation.

On the other hand, the appearance of all the good program analysis tools proposes new challenges for future CTF challenge designers. Firstly, most of the CTF generation works by far are expensive. An automated, systematic method is necessary to meet the high demand of benchmarks from program analysis tools. What's more, new CTF challenges should be able to add obstacles to these techniques and tools, so as to put forward better methods and solutions.

With the fact that CTF challenges and program analysis tools can complement each other and help each other forward, in this dissertation, we introduce our design method for automated generating CTF challenges. In addition, we propose a new dimension as CTF challenge complexities, which is the ability to resist or obstacle the running of program analysis tools, or referred to as "Program Analysis Tools Resistance".

The difficulty is that in addition to interposing the features that impede the phasing in the right places, it is necessary to make sure that our features do not have a clear pattern that can be identified, and then simply deleted. The future of CTF competitions will evolve to better resistance features as well as more powerful analyzing techniques.

Bibliography

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. "42 variability bugs in the linux kernel: a qualitative analysis". In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM. 2014, pp. 421–432.

[2] Masooda Bashir et al. "Cybersecurity competitions: The human angle". In: *IEEE Security & Privacy* 13.5 (2015), pp. 74–79.

[3] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based greybox fuzzing as markov chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 1032–1043.

[5] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *arXiv preprint arXiv:1803.01307* (2018).

[6] Ronald S Cheung et al. "Effectiveness of cybersecurity competitions". In: *Proceedings of the International Conference on Security and Management (SAM)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp). 2012, p. 1.

[7] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A platform for in-vivo multi-path analysis of software systems". In: *Acm Sigplan Notices* 46.3 (2011), pp. 265–278.

[8] DWARF Debugging Information Format Committee et al. "DWARF debugging information format, version 4". In: *Free Standards Group* (2010).

[9] DARPA. *Cyber Grand Challenge*. URL: https://http://archive.darpa.mil/cybergrandchallenge/.

[10] Brendan Dolan-Gavitt et al. "Lava: Large-scale automated vulnerability addition". In: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, pp. 110–121.

[11] Michael J Eager et al. "Introduction to the dwarf debugging format". In: *Group* (2007).

[12] Patrice Godefroid. "Random testing for security: blackbox vs. whitebox fuzzing". In: *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. ACM. 2007, pp. 1–1.

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.

[14] Istvan Haller et al. "Dowsing for overflows: a guided fuzzer to find buffer boundary violations." In: *USENIX Security Symposium*. 2013, pp. 49–64.

[15] Patrick Hulin et al. "AutoCTF: creating diverse pwnables via automated bug injection". In: *11th USENIX Workshop on Offensive Technologies WOOT 17)*. USENIX Association. 2017.

[16] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[17] George Klees et al. "Evaluating Fuzz Testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 2123–2138.

[18] Caroline Lemieux and Koushik Sen. "FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage". In: *arXiv preprint arXiv:1709.07101* (2017).

[19] Yuekang Li et al. "Steelix: program-state based binary fuzzing". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 627–637.

[20] Barton P Miller, Louis Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Communications of the ACM* 33.12 (1990), pp. 32–44.

[21] CWE MITRE. *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. URL: https://cwe.mitre.org/top25/.

[22] Gary Nilson et al. "BugBox: A Vulnerability Corpus for PHP Web Applications." In: *CSET*. 2013.

[23] NIST. *CVE-2015-3138 Detail*. URL: https://nvd.nist.gov/vuln/detail/CVE-2015-3138.

[24] NIST. *CVE-2017-9778 Detail*. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-9778.

[25] NIST. *CVE-2018-10534 Detail*. URL: https://nvd.nist.gov/vuln/detail/CVE-2018-10534.

[26] NIST. *Software Assurance Reference Dataset Test Suites*. URL: https://samate.nist.gov/SARD/testsuite.php.

[27] James Oakley and Sergey Bratus. "Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code." In: *WOOT*. 2011, pp. 91–102.

[28] PEACHTECH. *PEACH FUZZER COMMUNITY EDITION*. URL: http://www.peach.tech/resources/peachcommunity/.

[29] Jannik Pewny and Thorsten Holz. "EvilCoder: automated bug insertion". In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM. 2016, pp. 214–225.

[30] Sanjay Rawat et al. "Vuzzer: Application-aware evolutionary fuzzing". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2017.

[31] Jia Song and Jim Alves-Foss. "The DARPA Cyber Grand Challenge: A Competitor's Perspective". In: *IEEE Security & Privacy* 13.6 (2015), pp. 72–76.

[32] Nick Stephens et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution.” In: *NDSS*. Vol. 16. 2016, pp. 1–16.

[33] Ari Takanen, Jared D Demott, and Charles Miller. *Fuzzing for software security testing and quality assurance*. Artech House, 2008.

[34] Ruoyu Wang et al. “Ramblr: Making reassembly great again”. In: *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS17)*. 2017.

[35] Shuai Wang, Pei Wang, and Dinghao Wu. “Reassembleable Disassembling”. In: *USENIX Security Symposium*. 2015, pp. 627–642.

[36] Joseph Werther et al. “Experiences in Cyber Security Education: The MIT Lincoln Laboratory Capture-the-Flag Exercise.” In: *CSET*. 2011.

[37] Michal Zalewski. “American Fuzzy Lop (AFL)”. In: *December* (2016).