iut de BORDEAUX

University of BRISTOL

FUZZING BUG DEPTH EVALUATION

(How deep a bug is)

Internship realised by
Degail Dylan
From the 19th of April to the 25th of June

Supervisor :	**Sanjay Rawat**
Teacher responsible:	**Bruno Mery**
University year:	**2020 - 2021**

Abstract

Through the last years, Fuzzing is the most efficient technique to find vulnerabilities in real-world programs. AFL, the most popular fuzzer, is powerful to find a lot of bugs quickly. However, some bugs remain hidden, deeply in the binary. My research internship is to understand bugs and especially to know their “depth” into a binary. Intel Pin Tool, a dynamic binary instrumentation tool, can help me to analyze these bugs and get some relevant metrics. By counting conditional branches and call instructions, I can insight that, at a certain level, we have complex and deep bugs which can be really difficult to find. My goal will therefore be to, through my evaluation results, discuss possible solutions to understand them more and why not, catch them all!

Résumé

Au cours des dernières années, le Fuzzing est la technique la plus efficace pour trouver des vulnérabilités dans des programmes importants. AFL, le fuzzer le plus populaire, est très puissant et efficace pour trouver rapidement de nombreux bogues. Cependant, certains bougues restent cachés, profondément dans le binaire. Mon stage de recherche est de comprendre les bogues et surtout connaître leur "profondeur" dans un binaire. Intel Pin Tool, un outil d'instrumentation binaire dynamique, peut m'aider à analyser ces bogues et à obtenir des métriques pertinentes. En comptant les branches conditionnelles et les instructions d'appel, je peux en déduire qu'à un certain niveau, nous avons des bogues profonds et complexes qui peuvent être très difficiles à dénicher. Mon but sera donc de, à travers mes résultats, discuter autour de possibles solutions pour mieux les comprendre et pourquoi pas tous les attraper !

Acknowledgements

I would like to thank all of my teachers in the IT department for these two years and more particularly Ms Cartier, English teacher and Mr Mery, teacher and researcher in computer science for allowing me to obtain an internship abroad (even if there was the covid).

I would also like to thank Sanjay Rawat, my tutor who is a teacher and researcher at the University of Bristol in the cybersecurity research group, for letting me discover a new field and for having learned a lot of things about software analysis while implementing what I have learned over the past two years. He was very attentive, even when I couldn't express myself properly at times. He didn’t hesitate to repeat and give examples to improve my understanding of the subject. Thank you so much! I will be happy to keep in touch with you.

Figures

Here, we have all the figures I use in my report. If they have no source, it means that I have created it.

Introduction

Fuzzing [1] is an efficient software testing technique to discover new bugs by sending random data as inputs to software. The goal is to have some crashes, errors or memory leaks. Over the years, researchers search to improve it with new approaches to increase the *code coverage* [2] and potentially find more bugs. This main technique is getting a lot of attention in both academia and industry. But, nowadays, it's difficult to evaluate the main features of fuzzing and what kind of bug we found quickly or not. The question is: how hard are bugs to trigger?

So, the goal of my internship is to evaluate the "depth" of some bugs in some software to know if this bug is complex or not to reach. Why this big and not the other? With the help of my tutor, Sanjay Rawat, I must analyse software with specific inputs to get some valuable metrics of a bug found to bring some correlations between metrics and the difficulty to find a bug.

Firstly, I will present the context of my internship, the University of Bristol, and his cybersecurity research group. Afterwards, I will introduce the subject with an overview. Subsequently, I will develop on the different parts of my mission: the background (research and developing part) and the evaluation part. Finally, I will conclude with the results, some discussion about them and some possible improvements.

1. The context of the internship abroad

1.1. The presentation of the University of Bristol

The University of Bristol is one of the most popular and successful universities in the UK (top 9), ranked in the world's top 58 in the QS World University Rankings 2021 [3]. It has existed since 1876 and was before the University College, Bristol. It was the first higher education institution in England to admit women on an equal basis to men.

The University of Bristol is composed of six academic faculties: Arts, Engineering, Health Sciences, Life Sciences, Science and Social Sciences and Law. We will concentrate on the engineering academic faculty and particularly, the computer science department.

Figure 1: University of Bristol - guardian.com

1.2. The computer science department

The computer science of the University offers some undergraduate and postgraduate study. We can have a bachelor of science, master of engineering or master of science. In the last year of your bachelor, for example, you can choose courses to personalize your studies.

The department provides high-performance computing for a lot of people (students, postgraduate researchers, etc.).

Figure 2: High-performance computing - bristol.ac.uk

1.3. The Cyber Security group

The computer science department is composed of some research groups and one of them is the Cyber Security group.

Among the 5 persons of the core team, Sanjay Rawat, my tutor, is one of them. He is a researcher and lecturer in the computer science department. His main topics are static & dynamic security program analysis and automated software vulnerability analysis.

Figure3: Sanjay Rawat, my tutor, Researcher/Assistant professor - bristol.ac.uk

However, because of the pandemic world, I was not able to go to Bristol to do my internship. It's unfortunate but, let's stay positive, I have learned and discovered an interesting domain and at the same time, practised English.

1.4. The work introduction

So, remotely, I started my internship on the 19th of April and discovered the topic of the internship: Fuzzing Bug Depth Evaluation. With some dynamic binary instrumentation and test sets of binaries with inputs, my goal was to evaluate the bug depth. After this, show results with interesting metrics. I will develop this in the next section.

2. Overview

2.1. The progress in the search for software vulnerabilities

Fuzzing is one of the techniques which improve the search for software vulnerabilities. A lot of fuzzer were created and some benchmarks were created to compare them. Some of them are basic. They generate a lot of inputs to feed the binary concerned. We can categorize fuzzers: *generation-based* or *mutation-based* for the way of generating inputs, *dumb* or *smart* for the input structure, *white-* or *grey-* or *black-box* for the program structure [4].

For instance, *american fuzzy lop* (AFL*)* [5], one of the most popular fuzzer, is a smart mutation-based fuzzer. Meaning, AFL generates new inputs by slightly modifying a seed input (i.e., mutation), or by joining the first half of one input with the second half of another (i.e., splicing) [6]. About the understanding of the program structure, AFL is a grey-box fuzzer. Meaning, AFL leverages coverage-feedback to learn how to reach deeper into the program [6].

After many years, some community patches and upgrades of AFL allow AFL++ to be the most efficient fuzzer on average.

Figure 4: AFL logo (not animated) - wikipedia.org

Another fuzzer, *VUzzer* [7], tries to generate more efficient inputs and prioritize application parts to test. It tries to learn about how input can travel a deep path in the application to test the code's part that can be vulnerable. By mutating the input, Vuzzer can avoid error handlings and generate inputs valid enough to cross the common error detection, and test the real input usage to find bugs. Sanjay Rawat works on this fuzzer and Jean-Sebastien Bontemps, the old trainee of the IUT, had to improve it.

2.2. Where are the bugs?

Bugs can be everywhere in a binary. Some of them are shallow, on the surface of the code, and others are deeper. Some of them are easy to reach and others are complex to trigger. So, we have a lot of types of bugs.

Fuzzing didn't tell us about what kind of bugs fuzzers find. We didn't know if, for example, the 10 bugs the fuzzer found were shallow or not, easy or not to trigger.

In fact, AFL finds a lot of bugs in very little time but his code coverage is limited. The goal of *FairFuzz* [8] is to fix this problem. By focusing on *rare branches* (branches that few inputs reach), it can improve the code coverage and find potentially other bugs and particularly, deep bugs. The more we cover your code, the more we can find bugs and potentially, critical bugs.

Figure 5: Surface vs deep bugs - me

2.2. Time to find the bugs

About the time to find a bug, there is a question to think: when do I stop the fuzzing? For example, after 45 minutes of fuzzing and the first bug found, we decide to stop fuzzing because we think we won't find any other bugs after the one found around 5 minutes. What is the probability to find a potential bug in the next 15 minutes? It's the same on a larger timescale. After 6 months of running, should we stop? Maybe in the next ten days, we'll find a critical bug...

2.4. Cost of vulnerability discovery

Another point to think, it's the cost of vulnerability discovery. Relating to paperwork [9], discovering new vulnerabilities are exponentially cost. For non-deterministic (random decisions not based on the input bytes) fuzzers, if we have the same number of machines as the last evaluation, we can cover the same code linearly faster. However, for uncovering code, it's exponentially slower. To be linearly faster, we have to increase twice the number of machines. In other words, re-discovering the same vulnerabilities are cheap, but finding new vulnerabilities is expensive. But, we can't simply add more machines to find more vulnerabilities. The future solution is to develop smarter and more efficient fuzzers. So, even with a small performance in terms of power machine or vulnerabilities discovery, we can eventually have exponential growth.

2.5. Today

Today, with Fuzzing techniques, we have discovered a lot of bugs and among them, some dangerous bugs. Indeed, last year, for example, researchers had found some dangerous vulnerabilities which are contained in browsers such as Google Chrome (heap buffer overflow in FreeType library) [10]. These vulnerabilities existed years ago. They were exploitable. So, everybody could be hit by that. That is why it's vital to trigger deep bugs, increase the code coverage and optimize the fuzzing in terms of relevant bugs, times to find and cost.

3. Background

3.1. Introduction

In this part of the internship, I have to understand how to analyze a binary to get some information about it. I have some instructions, paperwork to understand the challenges and tools to start my research and development part.

3.2. Statically linked vs Dynamically linked library

First of all, it's essential to understand their differences. On one hand, a binary "statically linked" contains all the functions of the static library and the main executable. On the other hand, a binary "dynamically linked" contains only the main executable. If he wants to use a function of the dynamic library during the runtime, he searches it on the memory if it exists or loads it.

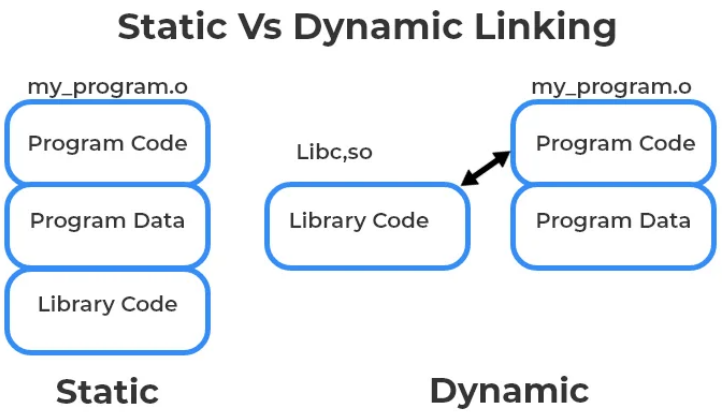

Figure 6: Static vs dynamic linking - prepinsta.com

About my analysis mission, a binary "statically linked" or "dynamically linked" gives us different results. Why? Image instrumentation allows me to concentrate on the main "executable". However, in the case of "statically linked", all of the functions of libraries are in the binary, so in our main "executable". Because of that, we instrument some parts of code that we don't want (functions of Libc for example).

In my case, in *Fuzzer Test Suite* [11], my test suite for the evaluate part (cf Evaluation setup), the bugs are in libraries. So, for one library, there is an utility, the main executable, which uses some functions about the library. After the build, there is a statically linked between the library and the utility. But, with an address range checking, I can control what I want to instrument (cf Appendices 2.1)

3.3. What is the “depth” of a bug?

A software has some components which can be relevant for the research:

- Branches.
- Functions.

These components can offer us a way to count the “depth” of a bug.

Firstly, branches in assembly language are the JMP (Jump) family instruction. In my case, I concentrated on the conditional branches (JMP with CMP (compare) instruction). It allows me to have my first metric to count the depth of a bug in a binary.

Secondly, functions in assembly language are the CALL instruction (and RET instruction when a function returns something). It allows me to have a *call graph* (a graph which represents calling relationships between subroutines in a computer program [12]) of the binary and to know the “depth” of a bug.

Here, we have a diagram to illustrate the second feature which use CALL instructions to know the “depth” of a bug:

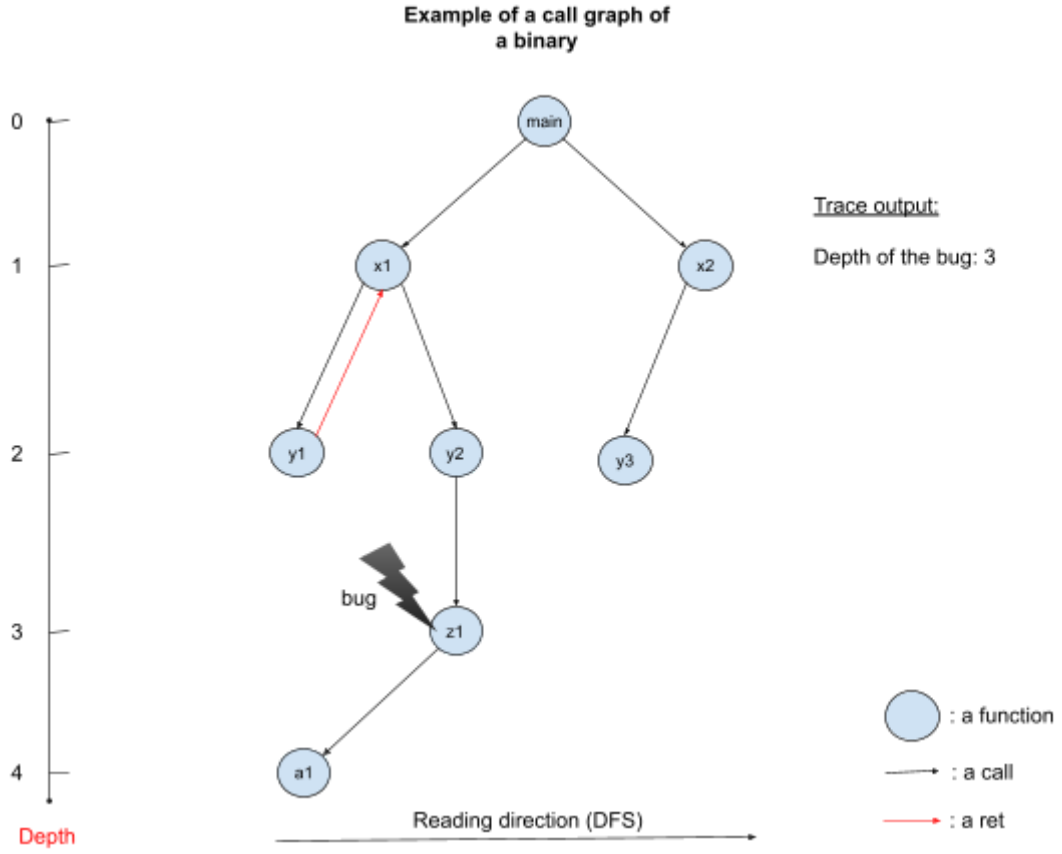

Figure 7: Example of a call graph of a binary - me

3.4. Dynamic binary instrumentation

The binary analysis involves analysing the object or executable code. We will concentrate on some part of the binary such as procedures, instructions, registers and so on.

The dynamic binary instrumentation uses this binary analysis to instrument a binary in runtime. It works by inserting code into a process to get some information about the runtime without changing the binary.

That was possible with the *Intel Pin* [13], a dynamic binary instrumentation tool that provides a rich and easy-to-use API to create scripts for security and especially software analysis. Pin allows you to observe the binary behaviour but not only. You can also change it by modifying instructions, register values, control flow and memory values.

To understand how it works, I have to introduce an important concept: *binary instrumentation* [14].

Binary instrumentation consists of two principles:

- The code you want to insert and where you want to insert it before the running of the binary. *It's the instrumentation*.
- The code to execute at insertion points during the instrumentation. *It's the analysis*.

To illustrate the first principle:

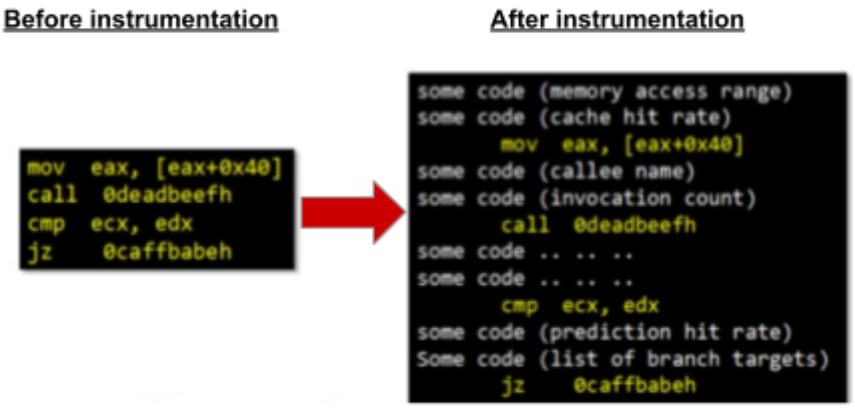

Figure 8: Example of binary instrumentation - ZongXian Shen and me

Example of code to execute (instructions counting):

```
// This function is called before every instruction is executed
VOID docount() { icount++; }
```

The analysis

```
// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
  // Insert a call to docount before every instruction, no arguments are passed
   INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}
```

The instrumentation

Figure 9: Example Pin script to count instructions - me

Another concept of instrumentation, we have different *instrumentation granularity* [15]. Here, we have a not limited list classified from general to precise instrumentation:

Image instrumentation: an IMG is all the data structures corresponding to a binary or loaded dynamic library.

↳ *Routine instrumentation*: a RTN represents functions/procedures in a binary.

↳ *Basic block instrumentation*: a BBL is composed of a sequence of instructions.

↳ *Instruction instrumentation*: an INS represents simply an instruction.

An image contains routines which contain basic blocks which contain instructions.

3.5. Conditional branch count

After having understood all of the concepts and understanding how to use Pin, I was able to start to create my custom Pin script.

Pin gives us some default/example scripts to start our custom Pin script. For example, we have a default script to count instructions of a binary. The principle is just to instrument every instruction of the binary and execute an incrementation count during the analysis. You have the piece of code on the last page.

A conditional branch count is based on the same principle of instruction count. We have to instrument a conditional branch, so a branch which can be taken if the condition is met. So I have to make a difference between the branches taken or not because, for my purpose, the “depth” of the bug is the number of conditional branches taken to reach it.

For each taken conditional branch, I get the complexity of the comparison for the conditional branch. I’ll explain it later.

Finally, I have two metrics with this feature:

- Number of conditional branches (no filter).
- Bug “depth” conditional branches (only taken branches).

So, my first job was to develop in C++ this feature with the Pin API by starting with the example Pin script.

3.6. CALL instruction count

After having integrated the first feature, the CALL instruction count was another cool and efficient metric to measure the “depth” of a bug and especially, *nested non-return CALL*.

This feature was certainly more difficult than the first one. I have to understand how a function is represented in low-level code.

Firstly, detect CALL and RET instructions is primordial. After that, I instrumented in different ways if it’s a CALL or RET instruction.

A thing that we have to think about is: how can we avoid function loops and even more, how can we avoid CALL instructions that call the same function in different places in the code? For the first part of the question, we can compare CALL instruction addresses and see if we have the same address or not. If there are the same, we are in a loop. For the second part of the question, the easy solution is to get the target address of a CALL instruction and do the same thing as the first. We have to avoid these cases because it can give us false results about the “depth” with CALL instructions.

Secondly, with the number of CALL instructions and RET instructions, I have only subtracted the two numbers to have only the nested non-return CALL (cf. Figure 7: Example of call graph of a binary).

Finally, I have three metrics with this feature:

- Number of call instructions (no filter).
- Number of unique functions (with the evoked cases solved).
- Bug “depth” call instructions (with the evoked cases solved and nested non-return CALL).

3.7. Complexity of conditional branches

The last feature is the complexity of conditional branches. It's a feature to complete the first one: *conditional branch count*.

Indeed, it helps me to know if a conditional branch is easy or complex to take.

The goal of this feature is to make a difference between comparison with 1-byte non-zero or multi-byte non-zero value. In fact, we can say that a comparison with a multi-byte non-zero value is more difficult to take than a comparison with a 1-byte non-zero value.

Firstly, I have to check the operand which has the value that I want to compare with only on the CMP and TEST instructions. In Intel syntax, it's the second operand.

After that, I have to treat the value of this second operand. It can be an immediate value or a register that can store comparison value.

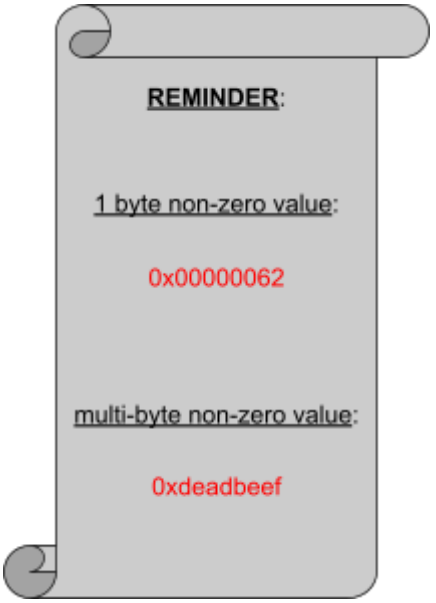

Figure 10: A reminder about byte value - me

Finally, I count the number of multi-byte non-zero values to know the complexity to reach a bug (by the complexity of the conditional branches in his way).

4. Evaluation

4.1. The evaluation process

4.1.1. Introduction

After the background (research and development part), we have to test our Pin script on a test suite to get some interesting results.

4.1.2. Manual review

Before that, to be sure that my script was correct, I have started with a manual review on a little script in C (cf Appendices 1) which can simulate crashes in different places on the code. I use the results and *Ghidra* [16], a software reverse engineering framework, to check if the results are logical with what I see on Ghidra. Everything seems correct to me.

```
[*] Main Binary Image: /root/Fuzzing-Bug-Depth-Evaluation-Internship/scripts/review_binaries/test-no-exit
[+] Image limits 0x94388672217088 - 0x94388672225703

ins address: 0x94388672221202 => branch count: 1 => taken count: 1 => complex?: false
ins address: 0x94388672221377 => branch count: 1 => taken count: 1 => complex?: false
ins address: 0x94388672221442 => branch count: 1 => taken count: 1 => complex?: false
ins address: 0x94388672221588 => branch count: 80400 => taken count: 80000 => complex?: false
ins address: 0x94388672221696 => branch count: 401 => taken count: 400 => complex?: true

call/ret ins address: 0x94388672221210 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94388672221352 - target: 0x0 => call count: 1 => ret count: 0 (call not taken into account)
call/ret ins address: 0x94388672221400 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94388672221464 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94388672221506 - target: 0x94388672221264 => call count: 1 => ret count: 0
call/ret ins address: 0x94388672221511 - target: 0x94388672221360 => call count: 1 => ret count: 0
call/ret ins address: 0x94388672221524 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94388672221594 - target: 0x0 => call count: 0 => ret count: 400
call/ret ins address: 0x94388672221609 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94388672221654 - target: 0x94388672221296 => call count: 1 => ret count: 0
call/ret ins address: 0x94388672221677 - target: 0x94388672221545 => call count: 400 => ret count: 0
call/ret ins address: 0x94388672221703 - target: 0x94388672221595 => call count: 1 => ret count: 0
call/ret ins address: 0x94388672221728 - target: 0x94388672221280 => call count: 1 => ret count: 0
call/ret ins address: 0x94388672221750 - target: 0x94388672221280 => call count: 1 => ret count: 0
call/ret ins address: 0x94388672221785 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94388672221836 - target: 0x94388672221184 => call count: 1 => ret count: 0
call/ret ins address: 0x94388672221865 - target: 0x0 => call count: 1 => ret count: 0 (call not taken into account)
call/ret ins address: 0x94388672221892 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94388672221924 - target: 0x0 => call count: 0 => ret count: 1

The bug depth with conditional branches = 80403
Number of all conditional branches = 80809
Complexity to reach the bug (in number) = 400
The bug depth with call instructions = 2
Number of unique functions = 7
Number of all call instructions = 407
```

Figure 11: Results on test script with test-no-exit - me

```
[*] Main Binary Image: /root/Fuzzing-Bug-Depth-Evaluation-Internship/scripts/review_binaries/test-exit-1
[+] Image limits 0x94064834600960 - 0x94064834609575

ins address: 0x94064834605074 => branch count: 1 => taken count: 1 => complex?: false
ins address: 0x94064834605281 => branch count: 1 => taken count: 1 => complex?: false
ins address: 0x94064834605346 => branch count: 1 => taken count: 1 => complex?: false
ins address: 0x94064834605492 => branch count: 80400 => taken count: 80000 => complex?: false
ins address: 0x94064834605603 => branch count: 401 => taken count: 400 => complex?: true

call/ret ins address: 0x94064834605082 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94064834605256 - target: 0x0 => call count: 1 => ret count: 0 (call not taken into account)
call/ret ins address: 0x94064834605304 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94064834605368 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94064834605410 - target: 0x94064834605152 => call count: 1 => ret count: 0
call/ret ins address: 0x94064834605415 - target: 0x94064834605264 => call count: 1 => ret count: 0
call/ret ins address: 0x94064834605428 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94064834605498 - target: 0x0 => call count: 0 => ret count: 400
call/ret ins address: 0x94064834605512 - target: 0x94064834605200 => call count: 1 => ret count: 0
call/ret ins address: 0x94064834605561 - target: 0x94064834605184 => call count: 1 => ret count: 0
call/ret ins address: 0x94064834605584 - target: 0x94064834605449 => call count: 400 => ret count: 0
call/ret ins address: 0x94064834605610 - target: 0x94064834605499 => call count: 1 => ret count: 0
call/ret ins address: 0x94064834605740 - target: 0x94064834605056 => call count: 1 => ret count: 0
call/ret ins address: 0x94064834605769 - target: 0x0 => call count: 1 => ret count: 0 (call not taken into account)
call/ret ins address: 0x94064834605796 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94064834605828 - target: 0x0 => call count: 0 => ret count: 1

The bug depth with conditional branches = 80403
Number of all conditional branches = 80809
Complexity to reach the bug (in number) = 400
The bug depth with call instructions = 3
Number of unique functions = 7
Number of all call instructions = 406
```

Figure 12: Results on test script with test-exit-1 - me

```
[*] Main Binary Image: /root/Fuzzing-Bug-Depth-Evaluation-Internship/scripts/review_binaries/test-exit-2
[+] Image limits 0x93866290454528 - 0x93866290463143

ins address: 0x93866290458642 => branch count: 1 => taken count: 1 => complex?: false
ins address: 0x93866290458849 => branch count: 1 => taken count: 1 => complex?: false
ins address: 0x93866290458914 => branch count: 1 => taken count: 1 => complex?: false
ins address: 0x93866290459147 => branch count: 1 => taken count: 1 => complex?: true

call/ret ins address: 0x93866290458650 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x93866290458824 - target: 0x0 => call count: 1 => ret count: 0 (call not taken into account)
call/ret ins address: 0x93866290458872 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x93866290458936 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x93866290458978 - target: 0x93866290458720 => call count: 1 => ret count: 0
call/ret ins address: 0x93866290458983 - target: 0x93866290458832 => call count: 1 => ret count: 0
call/ret ins address: 0x93866290458996 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x93866290459041 - target: 0x93866290458768 => call count: 1 => ret count: 0
call/ret ins address: 0x93866290459105 - target: 0x93866290458752 => call count: 1 => ret count: 0
call/ret ins address: 0x93866290459128 - target: 0x93866290459017 => call count: 1 => ret count: 0
call/ret ins address: 0x93866290459292 - target: 0x93866290458624 => call count: 1 => ret count: 0
call/ret ins address: 0x93866290459321 - target: 0x0 => call count: 1 => ret count: 0 (call not taken into account)
call/ret ins address: 0x93866290459348 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x93866290459380 - target: 0x0 => call count: 0 => ret count: 1

The bug depth with conditional branches = 4
Number of all conditional branches = 9
Complexity to reach the bug (in number) = 1
The bug depth with call instructions = 3
Number of unique functions = 6
Number of all call instructions = 6
```

Figure 13: Results on test script with test-exit-2 - me

```
[*] Main Binary Image: /root/Fuzzing-Bug-Depth-Evaluation-Internship/scripts/review_binaries/test-exit-3
[+] Image limits 0x94184733921280 - 0x94184733929895

ins address: 0x94184733925394 => branch count: 1 => taken count: 1 => complex?: false
ins address: 0x94184733925601 => branch count: 1 => taken count: 1 => complex?: false
ins address: 0x94184733925666 => branch count: 1 => taken count: 1 => complex?: false
ins address: 0x94184733925816 => branch count: 201 => taken count: 200 => complex?: false
ins address: 0x94184733925929 => branch count: 1 => taken count: 1 => complex?: true

call/ret ins address: 0x94184733925402 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94184733925576 - target: 0x0 => call count: 1 => ret count: 0 (call not taken into account)
call/ret ins address: 0x94184733925624 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94184733925688 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94184733925730 - target: 0x94184733925472 => call count: 1 => ret count: 0
call/ret ins address: 0x94184733925735 - target: 0x94184733925584 => call count: 1 => ret count: 0
call/ret ins address: 0x94184733925748 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94184733925823 - target: 0x94184733925520 => call count: 1 => ret count: 0
call/ret ins address: 0x94184733925887 - target: 0x94184733925504 => call count: 1 => ret count: 0
call/ret ins address: 0x94184733925910 - target: 0x94184733925769 => call count: 1 => ret count: 0
call/ret ins address: 0x94184733926076 - target: 0x94184733925376 => call count: 1 => ret count: 0
call/ret ins address: 0x94184733926105 - target: 0x0 => call count: 1 => ret count: 0 (call not taken into account)
call/ret ins address: 0x94184733926132 - target: 0x0 => call count: 0 => ret count: 1
call/ret ins address: 0x94184733926164 - target: 0x0 => call count: 0 => ret count: 1

The bug depth with conditional branches = 204
Number of all conditional branches = 210
Complexity to reach the bug (in number) = 1
The bug depth with call instructions = 3
Number of unique functions = 6
Number of all call instructions = 6
```

Figure 14: Results on test script with test-exit-3 - me

4.1.3. Evaluation setup

To test my Pin script and run the evaluation, I have prepared my setup.

First of all, I have downloaded a test suite: *Fuzzer Test Suite* on GitHub. There are more than 20 libraries with almost 45 bugs. We have different kinds of bugs: buffer-overflow, heap-overflow, use-after-free etc.

Afterwards, I have configured a Google Cloud virtual machine on *Google Cloud Platform* [17] to run Pin with my Pin script and the binary with its inputs. A virtual machine with 1 core processor and 16 Go of ram on a Debian machine.

4.2. Results

Each graphic represents the evaluation results in an appropriate form for the later explanations. We have graphics for conditional branches, the information complexity of conditional branches, call instructions and an image of test suite information.

About the last image, for each binary, we have a list of inputs for each bug. With that, we have the type of the bug, the time to find it and if we use a *seed* [18] or not to find the bug.

4.2.1 Graphics

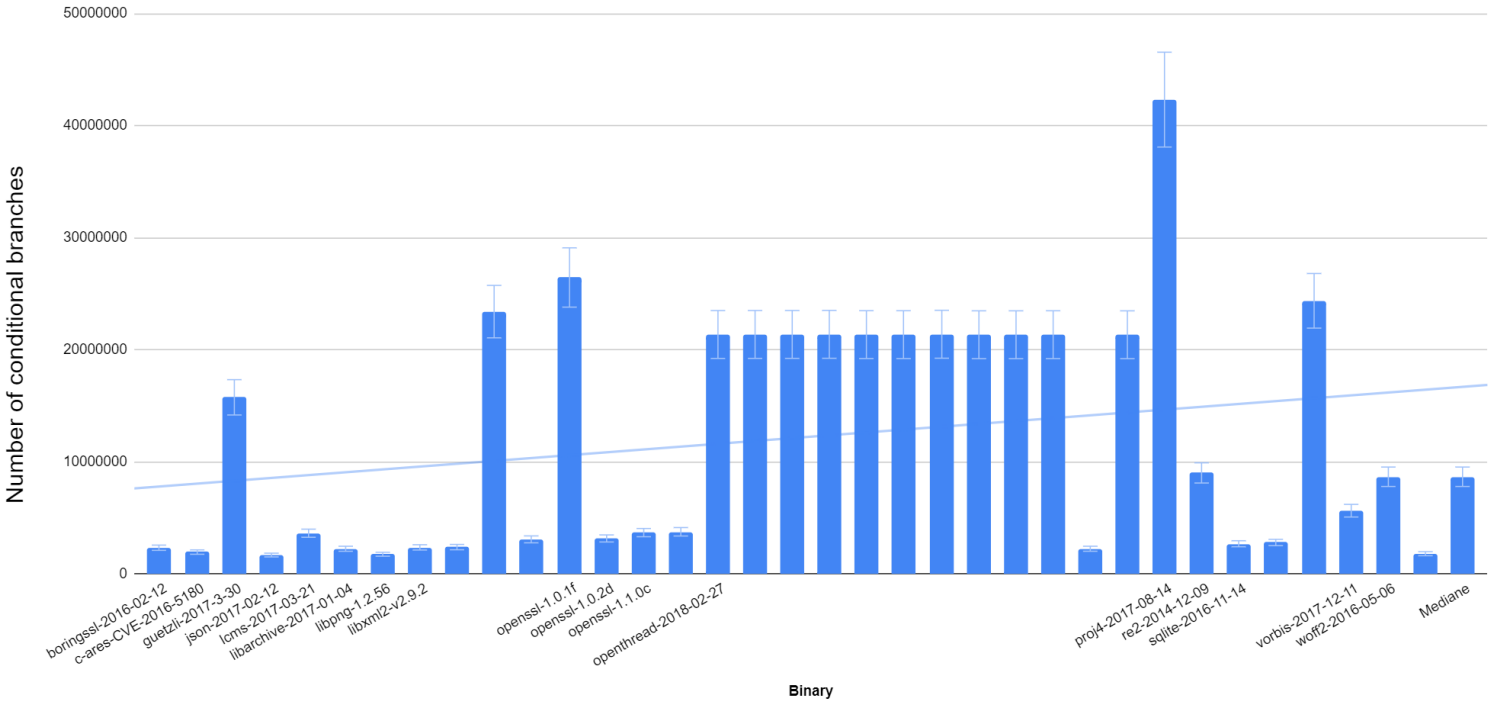

Figure 15: Number of conditional branches per binary - me

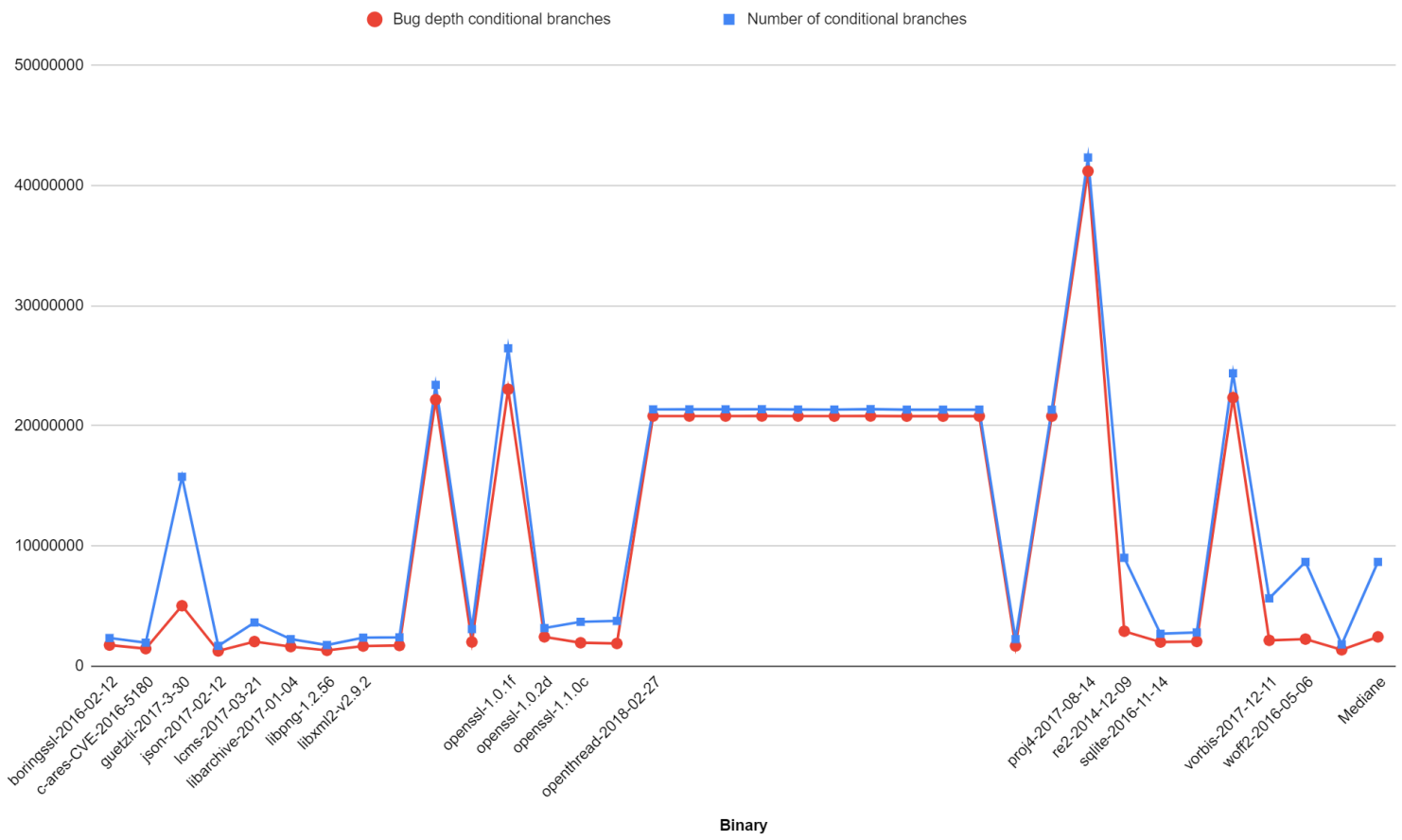

Figure 16: Number of conditional branches with bug depth per binary - me

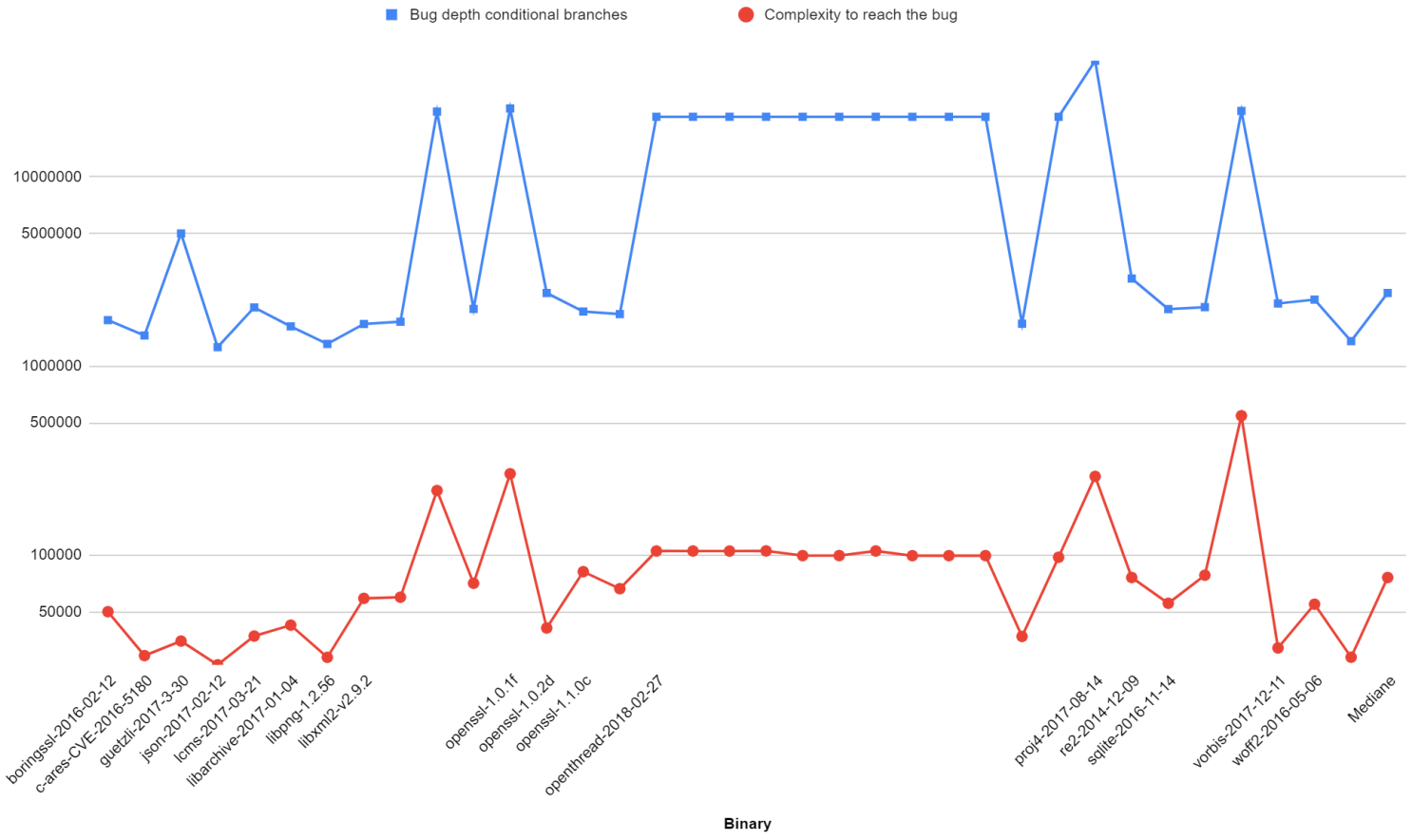

Figure 17: Bug depth conditional branches with complexity per binary - me

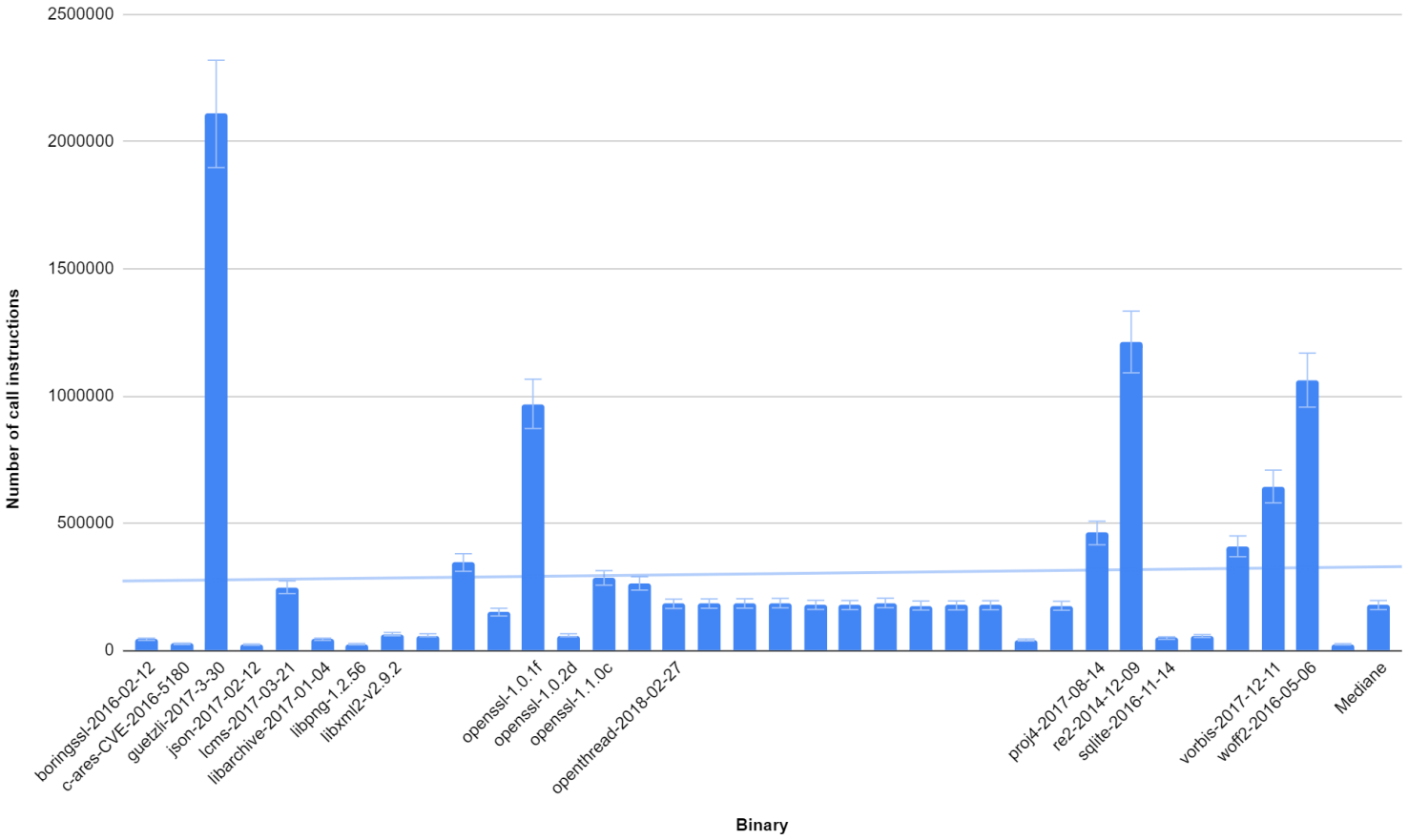

Figure 18: Number of call instructions per binary - me

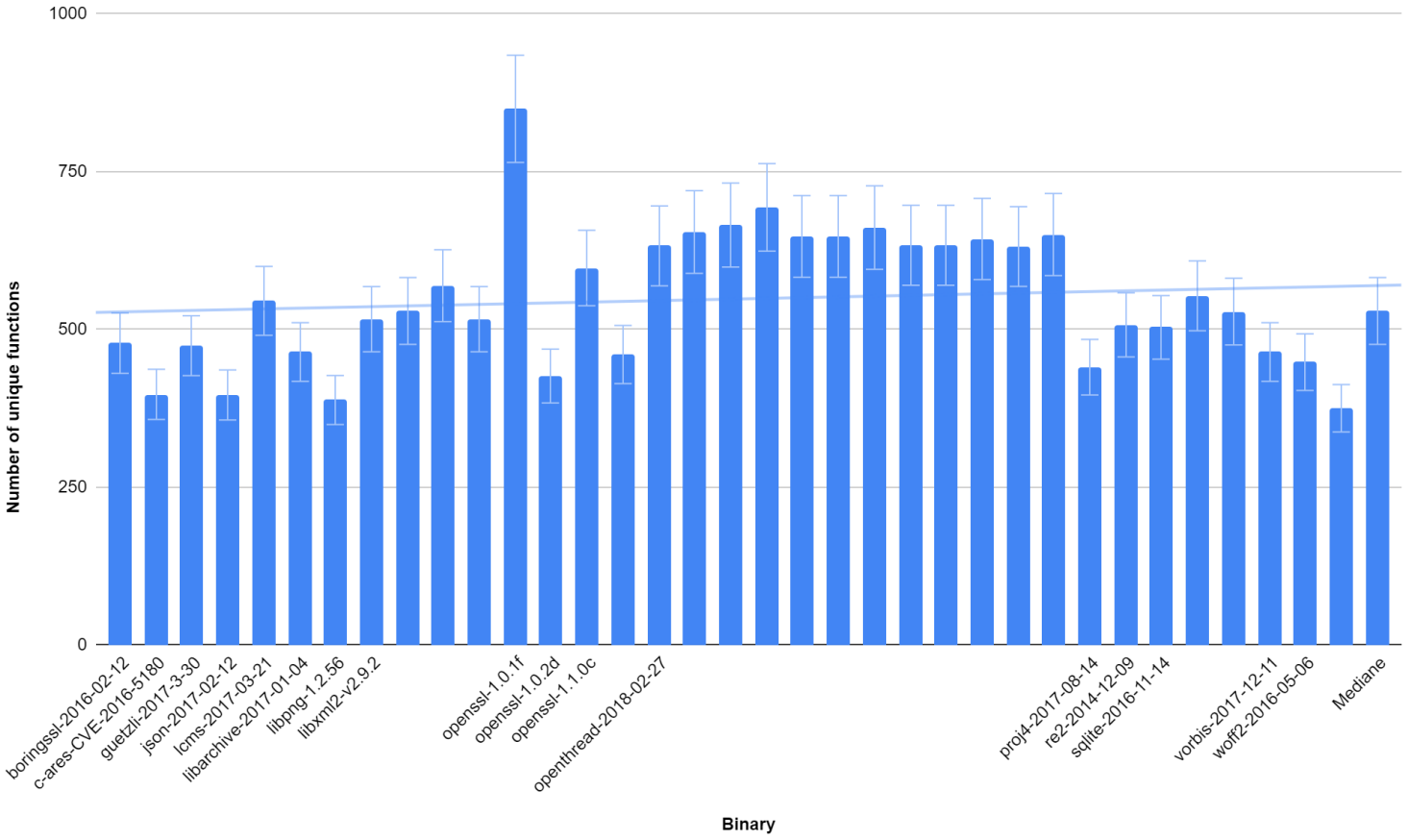

Figure 19: Number of unique functions per binary - me

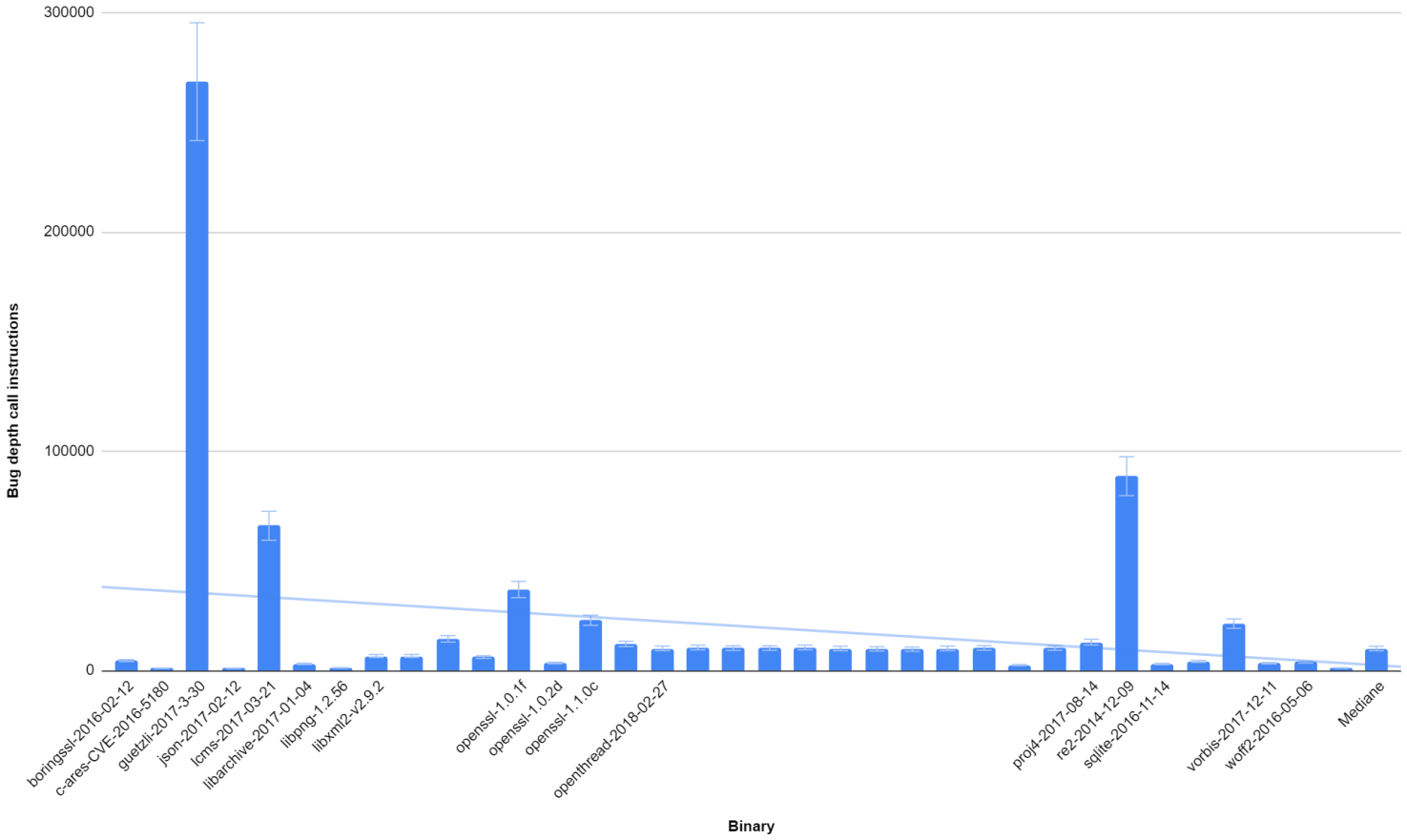

Figure 20: Bug depth call instructions per binary - me

Binary	Inputs	Type of bug (weight: 1)	Time to find (weight: 2)	Seeds (weight: 1)
boringssl-2016-02-12	crash-ffb22c3101db1e53e38fdf630efd3dfd19cbeb84	8-byte-read-heap-use-after-free	> 1 hour	No
c-ares-CVE-2016-5180	crash-36dde0db6beef00e19742fcfadda1df8358fb7f4 (others crash inputs but target same bug)	1-byte-write-heap-buffer-overflow	< 1 second	No
guetzli-2017-3-30	crash-5737651426557952	assertion failure	< 1 hour	Yes
json-2017-02-12	crash-1bd4fa4c5ddced4506c67dbe92e14ea7aa1bd47d	assertion failure	5 minutes	Yes
lcms-2017-03-21	crash-6a7f7b35fc6de5b19080b1c32588c727caf5d396	heap-buffer-overflow	30 minutes	Yes
libarchive-2017-01-04	clusterfuzz-testcase-minimized-6117891166437376	heap-buffer-overflow	1 hour	Yes
libpng-1.2.56	oom-63efa8b5a2adf76dc225d62939db3337ff6774f1 (reach a set of known source locations)	reach a set of known source locations	N/A	Yes
libxml2-v2.9.2	crash-50b12d37d6968a2cd9eb3665d158d9a2fb1f6e28	1-byte-read-heap-buffer-overflow	< 1 minute	No
	crash-d8960e21ca40ea5dc60ad655000842376d4178a1	1-byte-read-heap-buffer-overflow	N/A	No
	leak-bdbb2857b7a086f003db1c418e1d124181341fb1	memory leak	< 1 hour	No
	uaf-1153fbf466b9474e6e3c48c72e86a4726b449ef7	use-after-free	N/A	No
openssl-1.0.1f	leak-268f0e85f4bc45cbaf4d257222b830eac18977f3	multi-byte-read-heap-buffer-overflow (HeartBleed) memory leak	N/A	No
openssl-1.0.2d	crash-12ae1af0c82252420b5f780bc9ed48d3ba05109e	miscalculation miscalculation in OpenSSL's BN_mod_exp	< 1 minute	No
openssl-1.1.0c	crash-4fce1eeb339d851b72fedba895163ec1daab51f3 (fuzzer x509)	heap buffer overflow in X509v3_addr_get_afi	~9 months (and ~5 CPU years)	No
	crash-ab3eea077a07a1353f86eea4b6075df2e6319a75 (fuzzer bignum)	carry propagating in OpenSSL's BN_mod_exp	1 CPU year	No
openthread-2018-02-27	heap-94436b6f5f882f918e97ac74fb9a041375ab86b7 (repro1) (radio)	heap-buffer-overflow	N/A	No
	stack-49c80937be5de63c1c7b7652eacb22e1adc459b6 (repro2) (radio)	stack-buffer-overflow	N/A	No
	stack-71d40a5c838d345248fbc130c74182dda99d85f1 (repro3) (radio)	stack-buffer-overflow	N/A	No
	stack-71d40a5c838d345248fbc130c74182dda99d85f1 (repro4) (radio)	stack-buffer-overflow	N/A	No
	stack-ab4073980f120bbd4eb9f6d58950f2f03f88dac3 (repro5) (ip6)	stack-buffer-overflow	N/A	No
	stack-ab4073980f120bbd4eb9f6d58950f2f03f88dac3 (repro6) (ip6)	stack-buffer-overflow	N/A	No
	stack-7b706a9aa673042fa4586e19ab72c52769b493af (repro7) (radio)	stack-buffer-overflow	N/A	No
	stack-68a605f22e579ae45ab1d8221faa2d45e8668e05 (repro8) (ip6)	stack-buffer-overflow	N/A	No
	stack-68a605f22e579ae45ab1d8221faa2d45e8668e05 (repro9) (ip6)	stack-buffer-overflow	N/A	No
	stack-68a605f22e579ae45ab1d8221faa2d45e8668e05 (repro10) (ip6)	stack-buffer-overflow	N/A	No
	stack-bf52ed706facbbbd12b2d86c902c0f71b2b72bb0 (repro11) (ip6)	stack-buffer-overflow	N/A	No
	null-585080e3a0a1ee4287b9cb5745e470e6ac4c5c7b (repro12) (ip6)	null-dereference	N/A	No
proj4-2017-08-14	leak-7c19589a27e15f3432d245c7685bd518693e70d3	direct and an indirect leak	~ 2 and 10 minutes (if seeds are provided)	No
re2-2014-12-09	crash-a23ed2a04358b9c070c603a5af6ae2b34598664a	debug print and 8-byte-write-heap-buffer-overflow	< 10 seconds and < 1 hour	No
sqlite-2016-11-14	crash-0adc497ccfcc1a4d5e031b735c599df0cae3f4eb	heap-buffer-overflow	lots of CPU time	No
	crash-1066e42866aad3a04e6851dc494ad54bc31b9f78	heap-buffer-overflow	lots of CPU time	No
	leak-b0276985af5aa23c98d9abf33856ce069ef600e2	memory leak	N/A (more easy than buffer overflows)	No
vorbis-2017-12-11	crash-e86e0482b8d66f924e50e62f5d7cc36a0acb03a7	buffer overflow	several hundred CPU hours	Yes
woff2-2016-05-06	crash-696cb49b6d7f63e153a6605f00aceb0d7738971a	multi-byte-write-heap-buffer-overflow	< 20 minutes	Yes
	oom-9d24534a23b3ce397f21f62fb23ba9c5e9213107	hits OOMs	< 1 minute	No
Mediane				
Total binary: 17	**Total input: 35**			

Figure 21: Test suite information of Fuzzer-Test-Suite part 1 - me

Number of conditional branches (weight: 1)	Bug depth conditional branches (weight: 1.5)	Complexity to reach the bug (weight: 2)	Number of call instructions (weight: 1)	Bug depth call instructions (weight: 1.5)	Number of unique functions (weight: 1)
2331942	1747008	50054	43939	4601	478
1944292	1448716	29303	26397	1077	397
15755004	5021540	34989	2109175	268536	474
1679244	1256656	26201	22946	1026	396
3627296	2038936	37197	248504	66053	545
2240777	1618221	42478	43977	3140	464
1750460	1307421	28706	24735	1261	388
2363527	1666611	58871	64184	6695	516
2389777	1713121	59711	59658	6699	529
23404958	22169876	219079	345799	14518	569
3079584	2003193	70747	150755	6191	516
26451923	23039867	268802	969483	37003	849
3159842	2430713	41027	59684	3511	426
3681355	1939464	81528	285104	22980	597
3754843	1879805	66241	263590	12221	460
21364823	20810327	104858	183572	10142	632
21368074	20811769	104847	184319	10552	654
21369794	20810793	104807	184855	10267	665
21378816	20814668	104972	186089	10304	693
21352622	20807141	99096	179159	10573	647
21350915	20805556	99130	178494	10104	647
21386132	20815246	104907	186580	9918	661
21343222	20803324	99024	176578	9816	633
21343027	20802708	99060	177067	10158	633
21348042	20805463	99106	177722	10395	643
2242553	1671872	37028	40929	2547	631
21341402	20801499	97152	175768	10343	650
42328597	41195574	260266	461739	13025	440
9009719	2899236	75906	1212677	88671	507
2686849	1998311	55439	48962	3021	503
2799396	2046539	78078	56647	4212	553
24370162	22342106	544907	409417	21442	528
5637501	2139762	32174	644468	3391	464
8667496	2243419	54825	1062656	3907	448
1799766	1352393	28740	24369	1212	375
8667496	**2430713**	**75906**	**179159**	**10142**	**529**

Figure 22: Test suite information of Fuzzer-Test-Suite part 2 - me

4.2.2 Discussion

By viewing these graphics (Figure 15 to Figure 22), we can insight different interesting things.

First of all, about conditional branches, the median number of conditional branches of the bug depth is around 2.5 millions conditional branches to reach a bug. It's huge! The biggest one reaches 40 million conditional branches! About this, we can see that increase code coverage is really important to spread into the program and take a lot of conditional branches to find any bugs and maybe, critical bugs that are so deep.

Another thing, the complexity of the conditional branches gives us important information about how difficult it is to reach a bug. The median is around 75 thousand conditional branches which are complex to take. On average, we have 33% of complex conditional branches which can complicate the reach of a deep bug. That means that there is one-third of multi-byte non-zero values comparison and two-third of one-byte non-zero values comparison.

For example, if we take two binary:

- Openthread-2018-02-27 (average for all).
    - Conditional branches = around 21 million.
    - Bug depth conditional branches = around 20 million.
    - Complexity of conditional branches = around 100 thousand.

- SQLite-2016-11-14 (memory-leak).
    - Conditional branches = around 24 millions.
    - Bug depth conditional branches = around 22 millions.
    - Complexity of conditional branches = around 540 thousand.

We can see that, for the same conditional branches and bug depth, we can have a difference of complexity by 5. So, to reach the memory-leak of the SQLite program is more complex but not especially deeper than all of the bugs of the openthread program.

To conclude about conditional branches, we can say that a bug is complex to reach when you have a small ratio between bug depth conditional branches and complexity of conditional branches.

Secondly, about call instructions, the median number of call instructions is around 180 thousand call instructions. The biggest one is around 2.1 millions call instructions. It's so much! The median number of unique functions is 529 functions. About unique functions, we can see in figure 19 that we do not have very extreme values between binaries. It's maybe because they use the same base of popular functions such as printf(), malloc(), etc… And the difference is made with the specific functions of the binary.

Another thing, about bug depth call instructions, the median is around 10 thousand. It's a lot to reach a bug! The biggest one reaches around 270 thousand.

To finish, to see if the more a bug is complex, the more time it takes to find it, we can take the libxml2-v2.9.2:

- Libxml2-v2.9.2 (first crash 1-byte-read-heap-buffer-overflow).
    - Conditional branches = around 2,3 millions.
    - Bug depth conditional branches = around 1.7 million.
    - The complexity of conditional branches = around 60 thousand.
    - Call instructions = around 64 thousand.
    - Bug depth call instructions = around 7 thousand.
    - Unique functions = 516.
    - *Time to find = < 1 second.*
    - *Seed = No.*

- Libxml2-v2.9.2 (memory-leak).
    - Conditional branches = around 23 millions.
    - Bug depth conditional branches = around 22 millions.
    - Complexity of conditional branches = around 220 thousand.
    - Call instructions = around 350 thousand.
    - Bug depth call instructions = around 15 thousand.
    - Unique functions = 569.
    - *Time to find = < 1 hour.*
    - *Seed = No.*

We can see clearly in this example that the metrics *Time to find* and *Seed* are really important in the evaluation. There are simple metrics which can be useful to know if a bug is complex or not. The first bug is less complex than the second (in terms of metrics), so it takes less time to find it. But, if you have seed inputs (it's not the case in this example), it can accelerate the *Time to find*.

In the background part, I mentioned that when we run a fuzzer on binary, we don't know when we have to stop the fuzzer. In this example, we have a short *Time to find*, so it's not interesting. However, on a larger timescale, for example with Openssl-1.1.0c, it took ~9 months to find a bug. They would never have been able to find this bug if they had stopped after 8 months of research...

To conclude this discussion, I can say that, at a certain level, we can run dump or smart mutated-based generation inputs to find surface bugs as it does so well AFL fuzzer. When we reach a limited level and a code coverage that stagnates, we can run a coverage-guided or a targeted strategy fuzzer such as *LibFuzzer* [19] or FairFuzz to find the last complex and deep bugs of the program.

5. Conclusion

5.1. Possible improvements

5.1.1 Static binary instrumentation

For this evaluation, I used only one tool to instrument binaries: Intel PinTool. It's a tool to do dynamic binary instrumentation so instrumentation during runtime for example.

A possible improvement can be using a tool that allows me to do static binary instrumentation to compare and maybe find something between these two different methods.

Dyninst [20] is an example of static binary instrumentation.

5.1.2 Get more data and other features/metrics

Another possible improvement is getting more data to insight more easily or find a specific thing only if we have a lot of data.

We can also add other features than conditional branches and call instructions.

5.1.3 Refine some features/metrics

We can refine some metrics to have more precise results or to have more metrics by splitting a metric.

For example, about conditional branches, when I count the number of conditional branches, I didn't make the difference between a CMP/JMP instruction frequently called by a loop or by a function being called several times.

5.1.4 Reduce error rate during evaluation

For some metrics, I put an error rate because between two runs of evaluation, I can have a difference (not significantly of course). It can be linked with some scenarios that I didn't take into account.

5.2. Difficulties and limitations

The complicated thing about this internship was the limited time. My subject was so deep and vast that it can be a subject for a 6-months internship.

I had to learn a lot of things and understand the issues of today about Fuzzing which also reduced the time for evaluation.

5.3. Personal development

About technical knowledge, I have learned so much about software analysis and the structure of a binary. I learned how to use Pin, *Docker with Ubuntu image* [21] and Google Cloud Platform. I developed in C++ and C.

About human assets, I have practised oral English with my tutor and written English to write this report. I have learned to be professional and to be respectful about weekly meetings even if it was remotely.

5.4. End words

This internship made me discover a sub-domain of computer security so rich and so interesting which is binary analysis. It comforted me in the fact that I still like computer security, that there are still plenty of things to discover and that I intend to discover by continuing my studies through engineering studies and subsequently, specializing myself in computer security.

6. Appendices

1) The little script in C for the manual review:

```c
#include <stdio.h>
#include <string.h>

int foo() {
    int i, x = 0;

    // exit-2
    //exit(1);

    for (i = 0; i < 200; i++)
        x += i * 2;

    // exit-3
    //exit(1);

    return x;
}

int fii() {

    // exit-1
    //exit(1);

    return 1;
}

int main()
{
    int y, z = 0;

    const char* test = "a";
    //const char* test2 = "f";

    if (strcmp(test, "a") == 0) {

        for (int i = 0; i < 400; i++) {
            y += foo();
        }

        z = fii();
```

```
        printf("%d \n", y);
        printf("%d \n", z);
    } else {
        printf("%d \n", 0);
    }
}
```

2) The Pin script to analyse binaries (only interesting parts):

2.1) CheckBounds: the function to control what I want to instrument (in that case, the main executable)

```
// Checks if the instruction comes from the binary being instrumented.
BOOL CheckBounds(ADDRINT addr) {
    if(addr < IMG_HighAddress(MainBinary) && addr > IMG_LowAddress(MainBinary)){
        return true;
    }
    return false;
}
```

2.2) BranchCount: the analysis function for every conditional branches instrumentation

```
// This function is called before every branch is executed
VOID BranchCount(ADDRINT addr, BOOL taken)
{
    branchesCounter[addr]._branch++;
    if (taken)
        branchesCounter[addr]._taken++;
}
```

2.3) CallCount: the analysis function for every call instructions instrumentation

```
// This function is called before every call instruction is executed
VOID CallCount(ADDRINT addr, ADDRINT addrTarget, BOOL IsCall)
{

    std::pair<ADDRINT, ADDRINT> call (addr, addrTarget);

    if (IsCall) {

        callCounter[call]._call++;

    } else {

        callCounter[call]._ret++;
    }
}
```

2.4) ComplexityBranchCountREG: the analysis function for every conditional branch with a multi-bytes non-zero value comparison in register instrumentation

```
// This function is called before every we have a register in the CMP/TEST
instruction before a conditional branch
VOID ComplexityBranchCountREG(ADDRINT addr, CONTEXT *ctxt, string diss)
{
    // Get the name of the register
    std::string delimiter = ",";
    std::string token = diss.substr(diss.find(delimiter) + 2);

    // Reference to our registers reference dict
    for (int i = 0; !(regsRef[i].name.empty()); i++) {

        if (regsRef[i].name == token.c_str()){

            // Get the register value
            std::stringstream ss;
            ss << std::hex << std::setfill('0') << std::setw(8) <<
PIN_GetContextReg(ctxt, regsRef[i].ref);
            std::string reg_value = ss.str();

            int nb_non_zero_byte = 0;

            for (int i = 0; reg_value[i] != '\0'; i++) {
                if (reg_value[i] != '0') {
                    nb_non_zero_byte++;
                }
            }

            // About the complexity of the conditional branch
            if (nb_non_zero_byte <= 2) {
                branchesCounter[addr]._complexity = false;
            } else {
                branchesCounter[addr]._complexity = true;
            }

            nb_non_zero_byte = 0;
            break;
        }
    }
}
```

2.5) ComplexityBranchCountImmediate: the analysis function for every conditional branch information with a multi-bytes non-zero immediate value comparison instrumentation

```
// This function is called before every we have an immediate value in the CMP/TEST instruction before a conditional branch
VOID ComplexityBranchCountImmediate(ADDRINT addr, UINT64 value)
{
    // Get the immediate value
    std::stringstream ss;
    ss << std::hex << std::setfill('0') << std::setw(8) << value;
    std::string im_value = ss.str();

    int nb_non_zero_byte = 0;

    for (int i = 0; im_value[i] != '\0'; i++) {
        if (im_value[i] != '0') {
            nb_non_zero_byte++;
        }
    }

    // About the complexity of the conditional branch
    if (nb_non_zero_byte <= 2) {
        branchesCounter[addr]._complexity = false;
    } else {
        branchesCounter[addr]._complexity = true;
    }

    nb_non_zero_byte = 0;
}
```

2.6) Instruction: the instrumentation function for every instruction

```
// Pin calls this function for every instruction in the binary
VOID Instruction(INS ins, VOID *v)
{

    if (CheckBounds(INS_Address(ins))) {

        // Condition to insert a call (conditional branch and in the "main" executable)
        if (INS_IsBranch(ins) && INS_HasFallThrough(ins)) {

            // Insert a call to BranchCount before every branch
            INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)BranchCount,
                           IARG_INST_PTR, IARG_BRANCH_TAKEN,
                           IARG_END);

            // Get the last CMP or TEST instruction of the current conditional branch
            INS last_comparaison = INS_Prev(ins);
            while(INS_Invalid() != last_comparaison) {

                if (INS_Opcode(last_comparaison) == XED_ICLASS_CMP
                    || INS_Opcode(last_comparaison) == XED_ICLASS_TEST) {
                    break;
                }

                last_comparaison = INS_Prev(last_comparaison);
            }

            // Instrument CMP/TEST instruction for the complexity of the conditional branch
            if (INS_Valid(last_comparaison)) {

                if (INS_OperandIsReg(last_comparaison, 1)) {

                    INS_InsertCall(last_comparaison, IPOINT_BEFORE,
(AFUNPTR)ComplexityBranchCountREG,
                                   IARG_ADDRINT, INS_Address(ins),
                                   IARG_CONST_CONTEXT,
                                   IARG_PTR, new string(INS_Disassemble(last_comparaison)),
                                   IARG_END);

                } else if (INS_OperandIsImmediate(last_comparaison, 1)) {

                    int value = INS_OperandImmediate(last_comparaison, 1);

                    INS_InsertCall(last_comparaison, IPOINT_BEFORE,
(AFUNPTR)ComplexityBranchCountImmediate,
                                   IARG_ADDRINT, INS_Address(ins),
                                   IARG_UINT64, value,
                                   IARG_END);
```

```
                }
            }
        }

        // Condition to insert a call (call instruction or ret instruction)
        if (INS_IsCall(ins)) {

            if (INS_IsDirectControlFlow(ins)) {

                ADDRINT addrTarget = INS_DirectControlFlowTargetAddress(ins);
                INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)CallCount,
                               IARG_INST_PTR,
                               IARG_ADDRINT, addrTarget,
                               IARG_BOOL, true,
                               IARG_END);

            } else {
                INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)CallCount,
                               IARG_INST_PTR,
                               IARG_ADDRINT, 0,
                               IARG_BOOL, true,
                               IARG_END);
            }

        } else if (INS_IsRet(ins)) {

            if (INS_IsDirectControlFlow(ins)) {

                ADDRINT addrTarget = INS_DirectControlFlowTargetAddress(ins);
                INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)CallCount,
                               IARG_INST_PTR,
                               IARG_ADDRINT, addrTarget,
                               IARG_BOOL, false,
                               IARG_END);

            } else {
                INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)CallCount,
                               IARG_INST_PTR,
                               IARG_ADDRINT, 0,
                               IARG_BOOL, false,
                               IARG_END);
            }
        }
    }
}
```

2.7) Fini: the report function about the analysis

```
// This function is called when the application exits
// It closes the output file.
VOID Fini(INT32 code, VOID *v)
{

    for (std::map<ADDRINT, COUNTER_BRANCH>::iterator it=branchesCounter.begin();
it!=branchesCounter.end(); ++it) {
        if (it->second._taken != 0) {
            TraceFile << "ins address: 0x" << it->first
                      << " => branch count: " << it->second._branch
                      << " => taken count: "  << it->second._taken
                      << " => complex?: " << (it->second._complexity ? "true" :
"false") << "\n";

            // Only multi-bytes non-zero comparison
            if (it->second._complexity) {
                complexitycount += it->second._taken;
            }

           // Only conditional branches taken
            depthbranchcount += it->second._taken;
        }

       // All conditional branches (taken or not)
       branchcount += it->second._branch;
    }

for (std::map<std::pair<ADDRINT, ADDRINT>, COUNTER_CALL>::iterator
it=callCounter.begin(); it!=callCounter.end(); ++it) {
        if (it->first.second == 0 && it->second._call != 0) {
            TraceFile << "\n"
                      << "call/ret ins address: 0x" << it->first.first << " -
target: 0x" << it->first.second
                      << " => call count: " << it->second._call
                      << " => ret count: "  << it->second._ret
                      << " (call not taken into account)";

        } else {
            TraceFile << "\n"
                      << "call/ret ins address: 0x" << it->first.first << " -
target: 0x" << it->first.second
                      << " => call count: " << it->second._call
                      << " => ret count: "  << it->second._ret;
        }
```

```
        depthcallcount += it->second._call - it->second._ret;

        // Avoid function without target address
        if (it->first.second != 0 && std::find(targetAddressFunction.begin(),
targetAddressFunction.end(), it->first.second) == targetAddressFunction.end()) {
            targetAddressFunction.push_back(it->first.second);
            uniqcallcount++;
        }

        if (it->first.second != 0) {
            callcount += it->second._call;
        }

    }

    TraceFile << "\n" << "\n"
              << "The bug depth with conditional branches = " << depthbranchcount
<< "\n"
              << "Number of all conditional branches = " << branchcount << "\n"
              << "Complexity to reach the bug (in number) = " << complexitycount
<< "\n";

    TraceFile << "The bug depth with call instructions = " << depthcallcount <<
"\n"
              << "Number of unique functions = " << uniqcallcount << "\n"
              << "Number of all call instructions = " << callcount << "\n" <<
endl;

    if (TraceFile.is_open()) { TraceFile.close(); }
}
```

3) Bash script to control the evaluation:

```
#!/bin/bash

shopt -s nullglob
set -e

# arguments
binary=false
specific=false
if [ -z "${1}" ]
then
    echo "[+] Error: Please enter an application name"
    exit 1
fi

if [ -z "${2}" ]
then
    specific=true
    echo "[+] Info: Your application is in 'reach specific line' mode!"
fi

if [ -n "${3}" ]
then
    binary=true
    echo "[+] Info: Your application is in 'specific binary' mode!"
fi


# compile, build and run
DIR=$(find fuzzer-test-suite -type d -iname "${1}")
if [ -z "${DIR}" ]
then
    echo "[+] Error: Unknown application"
    exit 1
fi

NAME=$(echo "${DIR}" | cut -d "/" -f2)


# build script
BUILD=$(find scripts -type d -iname build)
if [ -z "${BUILD}" ]
then
    cd scripts && make && cd ..
fi
```

```
cd run_eval


# already run the binary?
BINARY=$(find . -type d -iname "${NAME}")
if [ -z "${BINARY}" ]
then
    mkdir "${NAME}" && cd "${NAME}" && ../../fuzzer-test-suite/"${NAME}"/build.sh
else
    cd "${NAME}"
fi


# run evaluation
if [ "${specific}" = true ] && [ "${binary}" = true ]
then
    ../../pin/pin -t ../../scripts/build/bugdepthevaluation.so -o
../../results/"${NAME}"-reach-line-bugdepthevaluation.out -- ./"${3}"

elif [ "${specific}" = true ] && [ "${binary}" = false ]
then
    ../../pin/pin -t ../../scripts/build/bugdepthevaluation.so -o
../../results/"${NAME}"-reach-line-bugdepthevaluation.out --
./"${NAME}"-fsanitize_fuzzer

elif [ "${specific}" = false ] && [ "${binary}" = true ]
then
    ../../pin/pin -t ../../scripts/build/bugdepthevaluation.so -o
../../results/"${NAME}"-"${2}"-bugdepthevaluation.out -- ./"${3}"
../../fuzzer-test-suite/"${NAME}"/"${2}"

else
    ../../pin/pin -t ../../scripts/build/bugdepthevaluation.so -o
../../results/"${NAME}"-"${2}"-bugdepthevaluation.out --
./"${NAME}"-fsanitize_fuzzer ../../fuzzer-test-suite/"${NAME}"/"${2}"
fi

cd ..
```

7. References

[1] Fuzzing - Wikipedia. https://en.wikipedia.org/wiki/Fuzzing.
[2] Code coverage - Wikipedia. https://en.wikipedia.org/wiki/Code_coverage.
[3] About the University of Bristol. https://www.bristol.ac.uk/university.
[4] Understand types of fuzzing techniques.
https://www.f-secure.com/en/consulting/our-thinking/15-minute-guide-to-fuzzing.
[5] American Fuzzy Lop - Wikipedia.
https://en.wikipedia.org/wiki/American_fuzzy_lop_(fuzzer).
[6] AFL and its characteristics. https://www.fuzzingbook.org/html/GreyboxFuzzer.html.
[7] VUzzer: Application-aware Evolutionary Fuzzing. https://github.com/vusec/vuzzer.
[8] FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage by Caroline Lemieux and Koushik Sen. https://www.carolemieux.com/fairfuzz-ase18.pdf.
[9] Fuzzing: On the Exponential Cost of Vulnerability Discovery by Marcel Böhme and Brandon Falk. https://mboehme.github.io/paper/FSE20.EmpiricalLaw.pdf.
[10] Google Chrome zero-day vulnerabilities.
https://www.tenable.com/blog/cve-2020-15999-cve-2020-17087-google-chrome-microsoft-windows-kernel-zero-day-vulnerabilities-exploited-in-wild-along-with-cve-2020-16009.
[11] Fuzzer Test Suite. https://github.com/google/fuzzer-test-suite.
[12] Call graph - Wikipedia. https://en.wikipedia.org/wiki/Call_graph.
[13] Intel Pin Tool: A Dynamic Binary Instrumentation Tool.
https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html.
[14] Explanation of instrumentation by Pin.
https://stackoverflow.com/questions/52686118/intel-pin-getting-instruction-memory-write-read-size.
[15] Introduction of Pin Tool instrumentation and instrumentation granularity.
https://slidetodoc.com/taint-analysis-contents-pin-tool-introduction-instrumentation-granularity/.
[16] Ghidra. https://ghidra-sre.org/.
[17] Google Cloud service. https://cloud.google.com/.
[18] What does seed input mean? https://techterms.com/definition/seed.
[19] LibFuzzer - library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.
[20] Dyninst. https://www.dyninst.org/dyninst.
[21] Docker and Ubuntu image. https://hub.docker.com/_/ubuntu.

Other good resources/related works:

Some examples of how to use Intel Pin Tool with examples:

- Malith Jayaweera | Bostonion Computer Scientist.
- JonathanSalwan/PinTools: Pintool example and PoC for dynamic binary analysis.
- Intel Pin Tool – Mustakimur Khandaker.
- shell-storm | Home.

GitHub repository for this internship:

- Fuzzing Bug Depth Evaluation Internship.