Design and Implement Binary Fuzzing based on LibFuzzer

Chun-Ying Huang, Wei-Chieh Chao, Si-Chen Lin, Yi-Hsien Chen
Department of Computer Science, College of Computer Science, National Chiao Tung University

***Abstract*—Libfuzzer is a coverage-guided fuzzing engine. But not like AFL, which can do the fuzzing without source code, Libfuzzer needs source code of the target to compile and do the fuzzing. This paper presents a way to use Libfuzzer to fuzz the binary without source code. Our implementation link the IO of binary with Libfuzzer, and use qemu to collect runtime information.**

I. Introduction

Fuzzing is a technique to automatically test software by providing random data as input. When fuzzing, we usually build a fuzzer to generate and send random inputs to the program. After program exit, it will send execution information back to the fuzzer. Then, the fuzzer will generate new input based on the execution information.

When doing fuzz testing, we focus on crashes. That is, we try to generate input data that make program crashed. Therefore, fuzzer analyzes execution information such as path, coverage, and exceptions to decide what kind of input data is next time.

American Fuzzy Lop (AFL) is one of the most famous fuzzing tools. There are two modes in AFL. One is source code mode, and the other is binary (QEMU) mode. Source code mode uses afl-gcc to compile the source code, in the meanwhile, it inserts lots of checkpoints to send execution information back. On the other hand, binary (QEMU) mode uses QEMU to run the binary, and it analyzes execution information from QEMU directly. Because of the implementation, its no doubt that source code mode would find crashes faster than binary (QEMU) mode.

LibFuzzer is a library for coverage-guided fuzz testing. It only support source code mode. Different from AFL, LibFuzzer considers target source code as library, and it send inputs through call function. It compile its source code with target source code together.

In some case, LibFuzzer found crashes much faster than AFL did. Heartbleed (CVE-2014-0160) was a critical security bug in the OpenSSL cryptography. It took couples of hours to find crash using AFL. However, it only took ten seconds to find crash using LibFuzzer. Thats one of the reasons that we wanted to build binary mode for LibFuzzer.

Generally, we built a bridge for communication between LibFuzzer and binary. We used QEMU, as same as AFL, to generate runtime information. In detail, we executed binary through QEMU and analyzed runtime information from QEMU after exited. LibFuzzer need lots of information from binary, such as coverage, caller, and cmp. We focused on coverage here. Coverage is how many percent of codes have been executed. First, we used radare2 to generate basic blocks infromation. Moreover, we did static analysis before executing binary. Second, we executed binary through QEMU and analyzed runtime information dynamically. Finally, we sent coverage infromation back to LibFuzzer.

II. Related Work

Sample citations [1], [2], [3], [4].

III. Methodology

A. Link IO together

Normally, when using Libfuzzer, we put the source code need to be test inside `LLVMFuzzerTestOneInput` as describe in the introduction section. But when we want to run a binary, we don't have source code. Hence, what we do is actually putting `execv("our-binary", ..., ...)` inside the `LLVMFuzzerTestOneInput` to invoke our binary. And before execute `execv` we need to use dup2 to link

1. fuzzer test input → binary stdin.
2. binary stdout → `/dev/null`.
3. binary stderr → `/dev/null`

The simplified code is something like below

```
extern "C" int LLVMFuzzerTestOneInput(const
    uint8_t *Data, size_t Size) {

    int P_IN[2]; pipe(P_IN);
    if(Size) write(P_IN[1], Data, Size);
    ...
    int pid = fork();
    if(pid == 0) {
        dup2(P_IN[0], STDIN_FILENO);
        dup2(dev_null, STDOUT_FILENO);
        dup2(dev_null, STDERR_FILENO);
        ...
        execv(..., ..., ...);
    }
    ...
}
```

B. Collect Runtime Information

After we link IO together, the binary should get the input of the libfuzzer test input now. Though it is runnable, Libufuzzer don't have any runtime information like code coverage and will stop running after a few rounds. First, we need to figure out what Libfuzzer collect while running. Libfuzzer collect runtime information through **Clang SanitizerCoverage**, which provides simple code coverage instrumentation and has hook function for customization. Now we use qemu to collect right information for those hook function implemented by LibFuzzer.

1) trace pc guard: LibFuzzer use two array to store the information of code coverage.

1. `__sancov_trace_pc_pcs` : store the program counter of the beginning of a code block.
2. `__sancov_trace_pc_guard_8bit_counters` : store how many times the code block was hit.

```
__sancov_trace_pc_pcs[Idx] = PC;
__sancov_trace_pc_guard_8bit_counters[Idx]++;
```

We need to fill those array for LibFuzzer to work. First, we use radare2 api **r2pipe** to do static analysis and index those code block. While doing static analysis with r2pipe, we filter out addresses we don't want, like library code. Second, we use qemu to check whether we hit a code block, and fill in those value. The qemu command we use is `qemu-x86_64 -D logfile -d in_asm binary`. `in_asm` provides us the assembly code and its address being executed. (see example below)

```
...
----------------
IN:
0x0000004000802c90:  mov    %r11,%rcx
0x0000004000802c93:  sub    %rax,%rcx
0x0000004000802c96:  cmp    $0xb,%rcx
0x0000004000802c9a:  ja     0x4000802dd8
...
```

After parsing qemu output and fill in those value, LibFuzzer is now runnable.

2) trace pc indir: Indirect call is a call instruction with nonconstant address. For example, `call rax`. Libfuzzer will gather information for indirect call using `__sanitizer_cov_trace_pc_indir` hook function. It needs the address before call, and the address after call. We implement this hook function in the following steps.

1. Modify qemu source code to output a text message when there is a indirect call.
2. When current translation block has a indirect call, we look at the ending address of the current block and the beginning address of the next block, then pass those values to `fuzzer::TPC.HandleCallerCallee`

IV. Evaluation

In this section, we will compare the performance of original LibFuzzer, our LibFuzzer and AFL. We use fuzzer-test-suite, a set of tests for fuzzing engines provided by Google, as the test cases. In order to support our LibFuzzer, we modify the `target.cc` in some test cases. Take `c-ares-CVE-2016-5180` for example, we modified the `extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)` function declaration to `int main()`. And use `read` function to read input data, the size is the length of input data. Check out more detail of patching fuzzer-test-suite in wiki page of our repository. The following specification is our machine information and all experiments were performed on it.

```
virtual private server on Google Cloud Platform
n1-standard-1 (one vCPU, 3.75 GB RAM)
```

A. Compared with original LibFuzzer

We choose some CVE test cases in fuzzer-test-suite, including `CVE-2016-5180`, `CVE-2015-8317` and `HeartBleed (CVE-2014-0160)`, to evaluate the performance of our LibFuzzer. These are the output of running test cases by our LibFuzzer and do it 10 times to calculate the average time.

1. `CVE-2016-5180`

This is a 1-byte-write-heap-buffer-overflow in c-ares. This bug was one of out a chain of two bugs that made a ChromeOS exploit possible: code execution in guest mode across reboots.

```
#0      READ units: 1
#2      INITED cov: 4 ft: 20 corp: 1/1b exec/s:
    2 rss: 25Mb
#3      NEW    cov: 4 ft: 39 corp: 2/3b exec/s:
    3 rss: 25Mb L: 2/2 MS: 1 CopyPart-
#4      pulse  cov: 4 ft: 59 corp: 2/3b exec/s:
    2 rss: 25Mb
#4      NEW    cov: 4 ft: 59 corp: 3/5b exec/s:
    2 rss: 25Mb L: 2/2 MS: 2 CopyPart-
    ShuffleBytes-
Error status : 139
Signal : Segmentation fault
```

And we can find the 1-byte-write-heap-buffer-overflow by our LibFuzzer within 3 seconds.

2. `CVE-2015-8317`

This is a 1-byte-read-heap-buffer-overflow and a memory leak in libxml2.

```
#0      READ units: 1
Error status : 139
Signal : Segmentation fault
```

Using our LibFuzzer can find the 1-byte-read-heap-buffer-overflow and a memory leak within 120 seconds.

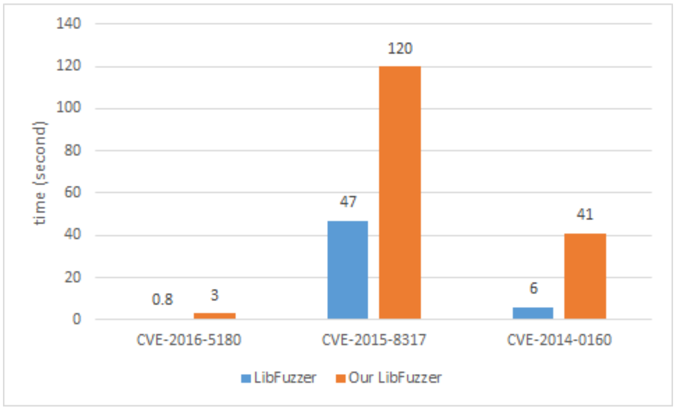
3. `HeartBleed (CVE-2014-0160)`

This is a multi-byte-read-heap-buffer-overflow in openssl.

```
#0      READ units: 1
...
 WARNING : block size exceeding max block size
     at 0x00457460
[+] Try changing it with e anal.bb.maxsize
 WARNING : block size exceeding max block size
     at 0x004f2ace
[+] Try changing it with e anal.bb.maxsize
Error status : 134
Signal : Aborted
```

In this test case, we cannot find the multi-byte-read-heap-buffer-overflow because this is a network service and there are a few asserts cannot pass by running program in local. As a result, we only got an aborted signal not segmentation fault.

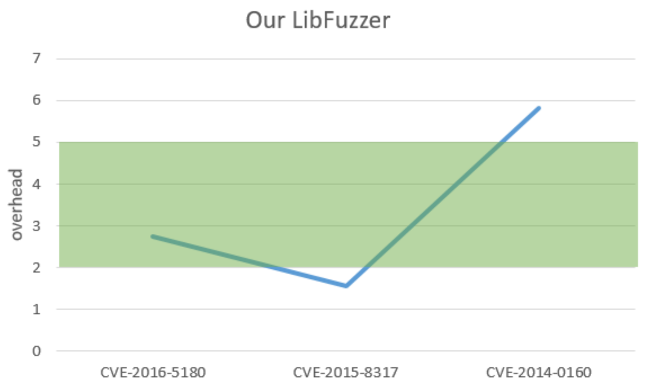
Compared with original LibFuzzer and plot the chart.



As we can see, because the implementaion of our LibFuzzer, the IO speed and analysis take a lot of time. And the preformance is slower than original LibFuzzer. But compared the time of original LibFuzzer and our LibFuzzer took, we still can find the bugs efficiently in these test cases.

B. Compared with AFL

According to the section of binary-only instrumentation in the technical whitepaper for afl-fuzz, we know the overhead of the QEMU mode is roughly 2-5x. So we try to compare with the overhead, the time of linking IO and analyzing the output, between AFL and our LibFuzzer. And the following chart is the overhead of the test cases we ran before.



The green area is the overhead of the QEMU mode and the bule line is our LibFuzzer overhead. We can see the result is close to the QEMU mode and therefore it maybe cost that much time to do analyze and link IO for supporting binary only mode.

V. Conclusion

We still got plenty of hook function we need to implement to make it works exactly like it should be.

References

[1] K. Serebryany, "libFuzzer a library for coverage-guided fuzz testing," LLVM project, 2015, https://llvm.org/docs/LibFuzzer.html.
[2] R. Swiecki, "honggfuzz," online, 2010, http://honggfuzz.com/.
[3] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2155–2168.
[4] K. Serebryany, "OSS-Fuzz - Google's continuous fuzzing service for open source software," 2017, USENIX Security. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany