Design and Implement Binary Fuzzing based on libFuzzer

Wei-Chieh Chao[1], Si-Chen Lin[1], Yi-Hsien Chen[1], Chin-Wei Tien[2], Chun-Ying Huang[1]
[1] Department of Computer Science, College of Computer Science, National Chiao Tung University
[2] Cybersecurity Technology Institute, Institute for Information Industry

***Abstract*—We design and implement libFuzzer-bin, a coverage-guided binary fuzzer based on libFuzzer. We discuss how libFuzzer-bin is implemented, measure the overheads of our implementation, and compare its performance against the American Fuzzy Lop (AFL) fuzzer. This is a work-in-progress. Our evaluation results show that, compared with the vanilla AFL, the current implementation brings only limited overheads and is able to identify possible vulnerabilities in a shorter time.**

I. Introduction

Fuzzing is a common technique used to perform automatic software testing. It starts with a user-provided initial input. A number of mutations are performed against the initial input to generate more test inputs. Test inputs are then used for fuzzing and testing if any unexpected behavior, e.g., crash, can be triggered by the program. A tool used to perform fuzzing test is often called a fuzzer. One popular class of fuzzers is grey-box fuzzers. Grey-box fuzzers collect runtime states from a program under testing, and it, therefore, achieves a balance between efficiency and effectiveness, compared to white-box and black-box fuzzers.

Most grey-box fuzzers require source codes to perform a fuzzing test. This is because the most efficient approach to collect program runtime states is to perform static instrumentation when compiling a program. Codes used to collect runtime states are embedded into a compiled program so that runtime states can be reported to the fuzzers in a fuzzing process. However, source codes may be not accessible for security engineers, and it shows the demand on grey-box based fuzzing for binaries.

To our knowledge, the American Fuzzy Lop (AFL) [1] is the only one grey-box fuzzer that support binary fuzzing. However, recent grey-box fuzzers such as libFuzzer [2] find crashes faster than AFL in several cases. It might because libFuzzer considers more runtime states than AFL. Currently, libFuzzer still has to work with source codes. Therefore, in this paper, we attempt to design and implement a binary fuzzer based libFuzzer and see what would be the critical factor to affect the performance of fuzzing.

II. Methodology

A. Overview

Before introducing how libFuzzer-bin is designed and implemented, we briefly introduce how libFuzzer is used to test a program. libFuzzer is designed for fuzzing source codes. The source codes must be compiled with `clang` and enable the required compiler hooks. This is because libFuzzer has to collect program runtime states via the hooks injected into the generated object codes. Readers, please refer to Section II-C for more details on compiler hooks of original LibFuzzer. In addition, enable the compiler hooks, a developer has to implement a function for receiving test inputs generated by libFuzzer. One function for receiving test inputs is the `LLVMFuzzerTestOneInput` function. The prototype for this function is defined as follows.

```
int LLVMFuzzerTestOneInput(const uint8_t *data size_t size);
```

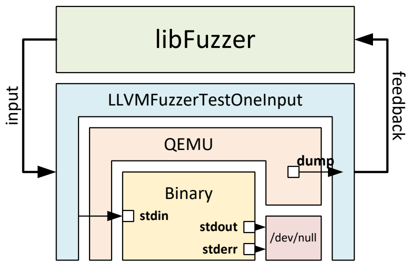

Fig. 1. The architecture of our design and implementation.

The `data` and the `size` parameters are used to pass test inputs generated by libFuzzer. Functions to be fuzzed are called inside the `LLVMFuzzerTestOneInput` function. Since the functions are compiled with injected hooks, the runtime states are collected and updated when a hook is triggered.

In this work, we aim to perform fuzzing against binary executables instead of source codes. Therefore, a number of challenges must be addressed before it can work properly. In addition, maintain an acceptable performance, challenges including how the hooks are identified and how runtime states are collected are addressed in the following subsections.

B. Architecture

The architecture of libFuzzer-bin is illustrated in Figure 1. libFuzzer-bin is implemented as a standalone program and is linked against libFuzzer. As a result, libFuzzer-bin contains the "libFuzzer " and "LLVMFuzzerTestOneInput" parts shown in the figure. To simplify our implementation and maintain program tracing performance, we leverage the QEMU [3] program emulator. libFuzzer-bin launches a program to be fuzzed (the "Binary" in the figure) using QEMU user mode. It is worth noting that we disabled the output from the binary launched by QEMU, and patched the QEMU to send required runtime traces to libFuzzer-bin. Once we have done the setup and run the program, test inputs fed by libFuzzer can be forwarded to the program under test via its standard input. Program runtime traces reported from QEMU are then used to update internal states required by libFuzzer.

C. LLVM Hooks

libFuzzer leverages LLVM hooks to collect and update program runtime states. To be more specific, the important runtime states collected via hooks includes 1) coverage, 2) indirect function calls, 3) memory search, 4) comparison, and 5) address sanitizer [4]. For the details about the hooks, readers can refer to LLVM and libFuzzer documents. In a fuzzing process, users can choose which states should be collected and used to direct the fuzzing process. However, the coverage counter is the minimum requirement and is used by most grey-box fuzzers. libFuzzer-bin is a work-in-progress and currently, only the first three items are collected.

D. Runtime State Collection

We introduce how runtime state collection and libFuzzer libFuzzer-state updating are implemented in libFuzzer-bin. We receive runtime trace dumps from QEMU user mode, parse the output, and update states required by libFuzzer. Currently, we update runtime states in terms of coverage, indirect calls, and memory searches.

a) Coverage: libFuzzer uses two arrays to store runtime code coverage, i.e., the `__sancov_trace_pc_pcs` and the `__sancov_trace_pc_guard_8bit_counters` arrays. The former stores the program counter of the beginning of a code block, and the latter stores how many times a code block is visited. To simplify the identification of basic blocks, we leverage the radare2 api `r2pipe` to perform static analysis and index basic blocks. With the identified basic blocks within a program under testing, libFuzzer-bin focuses only on fuzzing the program and skip code segments that are irrelevant to the program, e.g., library codes.

b) Indirect Calls: An indirect call is a function call to a non-constant function address. For example, function call based on a function pointer is an indirection call. The address of a target function can be placed in either a register or a memory address. libFuzzer collect indirect calls by using the `__sanitizer_cov_trace_pc_indir` hook function. This function requires the address before a call and the address after the call. To collect the states, we patched QEMU to determine if the end of a basic block is a `call` instruction and the address of the called target. If the target is a register or a memory address, we pass the end address of the current block and the beginning address of the next block to the hook function.

c) Memory search: libFuzzer also collect the use of memory search relevant functions such as `memmem` and `strstr`. Identifying function calls to interesting functions are similar to identifying indirection calls. The only difference is that libFuzzer-bin has to determine if a target function address is an interested function. This is simple for dynamically linked objects because we would be able to determine the functions from the `plt` table. However, it could be difficult if the program is statically linked against the C library.

III. Evaluation

We further compare the performance of libFuzzer-bin against the original libFuzzer and the popular grey-box fuzzer AFL [1]. We select test cases from fuzzer-test-suite [5] as the test cases, which a set of test cases for fuzzing engines provided by Google. The selected test cases are `CVE-2016-5180`, `CVE-2015-8317` and `HeartBleed (CVE-2014-0160)`. To work the libFuzzer-bin, we slightly modify the test cases if they do not implement a `main` function but only implement the `LLVMFuzzerTestOneInput` function. An additional `main` function is added to the test case to read test inputs from standard input and pass read inputs via the `LLVMFuzzerTestOneInput` function. We perform the evaluations on a virtual server hosted on Google cloud platform. The virtual server use the `n1-standard-1` configuration (single vCPU, 3.75GB RAM).

A. Correctness

We evaluate the correctness of libFuzzer-bin by comparing the results with those from the original libFuzzer. Compared to the original libFuzzer, we can successfully find the vulnerabilities from the first two test cases. Although libFuzzer-bin is able to generate inputs that crash the test case, it fails to find the vulnerability because of the lack of address sanitizer hooks. Figure 2 shows the

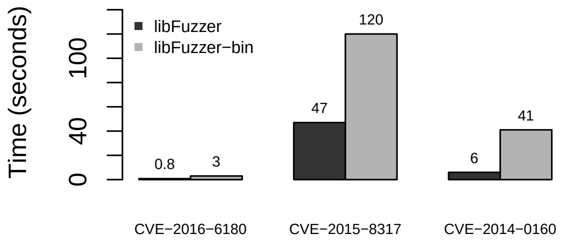

Fig. 2. Performance Comparison for native libFuzzer and libFuzzer-bin.

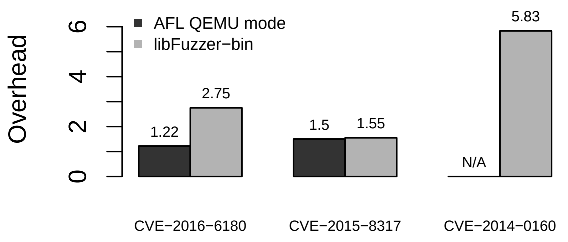

Fig. 3. Performance comparison against AFL.

performance numbers for the three test cases. It shows that the dynamical instrumentations performed by libFuzzer-bin is about 3x to 6x slower than static instrumentation in native programs.

B. Performance

We also compare the performance of libFuzzer-bin against that of AFL QEMU mode, which also works for fuzzing binaries. According to the statements from AFL document, running AFL in QEMU mode incurs roughly 2x to 5x overheads compared to that without instrumentation. We measure the performance overhead by comparing the running time of a program with and without instrumentation. Figure 3 shows the overheads measured from libFuzzer-bin. It shows that our design and implementation is efficient and competitive to that of AFL. Note that AFL fails on running the test cases for CVE-2014-0160, so an "N/A" is marked on that bar.

IV. Conclusion

We design and implement libFuzzer-bin, a binary fuzzer based on libFuzzer. This is a work-in-progress. Although not all hooks required by libFuzzer are implemented, our preliminary evaluations show that the current libFuzzer-bin is effectiveness and efficiency. Our future work is to design and implement all libFuzzer required hooks and conduct intensive experiments to show its effectiveness.

References

[1] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.
[2] K. Serebryany, "libFuzzer a library for coverage-guided fuzz testing," LLVM project, 2015, https://llvm.org/docs/LibFuzzer.html.
[3] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track*.
[4] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Presented as part of the 2012 USENIX Annual Technical Conference*. USENIX, 2012, pp. 309–318.
[5] Google Inc., "fuzzer-test-suite," https://github.com/google/fuzzer-test-suite.