Fuzz Testing in Practice: Obstacles and Solutions

Jie Liang*, Mingzhe Wang*, Yuanliang Chen* Yu Jiang*‡ and Renwei Zhang†
* School of Software, Tsinghua University, KLISS, Beijing, China
† Huawei Beijing Research Institute, Beijing, China
Email: {liangjie.mailbox.cn, wmzhere}@gmail.com, chenyuan17@mails.tsinghua.edu.cn,
jiangyu198964@126.com, zhangrenwei1@huawei.com

***Abstract*—Fuzz testing has helped security researchers and organizations discover a large number of vulnerabilities. Although it is efficient and widely used in industry, hardly any empirical studies and experience exist on the customization of fuzzers to real industrial projects. In this paper, collaborating with the engineers from Huawei, we present the practice of adapting fuzz testing to a proprietary message middleware named `libmsg`, which is responsible for the message transfer of the entire distributed system department. We present the main obstacles coming across in applying an efficient fuzzer to `libmsg`, including system configuration inconsistency, system build complexity, fuzzing driver absence. The solutions for those typical obstacles are also provided. For example, for the most difficult and expensive obstacle of writing fuzzing drivers, we present a low-cost approach by converting existing sample code snippets into fuzzing drivers. After overcoming those obstacles, we can effectively identify software bugs, and report 9 previously unknown vulnerabilities, including flaws that lead to denial of service or system crash.**

I. Introduction

Coverage-based greybox fuzzers use coverage information to direct the generation of random inputs, then run the program with the input and catch abnormal behavior. They have helped identify a large number of software bugs and security vulnerabilities. Attracted by the effectiveness, more and more open-source organizations and software companies are adopting fuzz testing to their projects [5, 1, 6]. For example, Google started project OSS-Fuzz [10], which offers fuzz-as-a-service for open-source organizations.

While fuzz testing is spreading in industrial practice, academic researchers are also improving fuzzers in various ways. Most researchers focus on the efficiency of seed scheduler and mutation- or generation-based algorithms. Complex components such as symbolic executor [4] and static analyzer [12] are also introduced, to further improve the traditional greybox fuzzers, like American Fuzzy Lop [15] and libFuzzer [8].

For simplicity and availability, researchers usually pick modular open-source projects and standard benchmarks to evaluate their works. For example, Google's fuzzer-test-suite [11] is intended to evaluate the performance of libFuzzer, where almost all tests in the suite are protocol-related or file-parsing libraries with few dependencies. AFLFast [2] is evaluated with GNU binutils, a set of single-threaded binary utilities with only one dependency. Driller [13] is evaluated with Cyber Grand Challenge's binaries, where programs, libraries, even the compiler and the runtime are specially crafted.

‡Yu Jiang is the correspondence author, and the work is supported by Tsinghua-Huawei Collabration Project YBN2017010031.

Most academic tools show good performance in simplified environment. However, in practical environment, when we apply those tools to real industrial projects, they do not work as well as what the benchmarks indicate. Too many causes may fail a "smart" fuzzer, due to the increasing complexity in system designs and execution environment dynamics. For example, two testing engineers from Huawei spent two weeks trying to run Driller on `libmsg`, a messaging middleware. With the source code of both projects in hand, unfortunately, they still failed to run the fuzz testing and learned painful lessons: Driller didn't support a set of POSIX APIs and had rigid requirements on system and library configuration.

In this paper, we collaborate with the engineers from Huawei to adapt the fuzzing techniques to `libmsg` and identify several typical obstacles coming across through the engineering practice. The main obstacles include system configuration inconsistency, system build complexity, fuzzing driver absence, fuzzing performance degradation, and bug-hiding code segments. For each obstacle, our solution is also provided and some supporting tool modules are developed. For example, we develop a tool module to mitigate the system build complexity. We also propose a method that mitigates the absence of fuzzing drivers, i.e. converting existing unit tests and sample code to fuzzing drivers. By providing solutions to those obstacles, increased acceptance and application of fuzzing tools in the industry can be achieved.

Finally, after overcoming those obstacles, we run the fuzz testing tool SAFL [14] successfully and effectively identify several software bugs. 9 previously unknown vulnerabilities are reported, including flaws that lead to slow resource leak, denial of service or immediate system crash. Except for those detected vulnerabilities, we also share a set of interesting observations regarding different stages of fuzz testing. For example, developers may write code weakening fuzzer's ability to detect anomalies.

II. Background

A. Messaging Library `libmsg`

Originating from project M, `libmsg` has been operating reliably for nearly 10 years. As the core messaging library, and all microservices in Project M depend on it. Recently, an evolved version of `libmsg` is rolling out in Huawei. The major changes include the following two aspects:

- Component separation. `libmsg` is separated from Project M to promote the use of `libmsg` to a wider range of products and finally support message transfer

of the entire distributed system department. To fulfill the dependencies of `libmsg`, the separation requires major changes to the build system.
- Protocol optimization. Originally, `libmsg` mainly uses HTTP as the underlying protocol for cross-platform and cross-language support. The rework adds binary protocol msgio, which is faster but lack of platform-independence. To reuse existing library code and expose the same interface to clients while implementing one more protocol, many components' logic is changed accordingly.

Unexpectedly, the system is found unstable after the upgrade. We work closely with the development team, aiming to pinpoint bugs with fuzz testing techniques.

B. Fuzz Testing Tool SAFL

Fuzz testing is considered as one of the most efficient approaches to find security vulnerabilities in software systems. The core idea is to generate and feed the software systems with random inputs, then capture abnormal behaviors such as program crash. Coverage-based greybox fuzzing technique improves blackbox fuzzing by adding feedback of coverage information. Greybox fuzzers can use this information to focus on new and interesting paths and discard explored ones.

The fuzzer used in this paper is SAFL, our most recently developed fuzz testing toolkit. The algorithm behind is a combination of FairFuzz [7] and AFLFast [2]. The idea of FairFuzz is to overcome the unfair seed selection. Only a small amount of seeds cover rare branches, and they are neglected by the default strategy of AFL. FairFuzz fixes the problem by taking branch rarity into account. The idea of AFLFast is to overcome the slow seed mutation. A seed exercising rare path has more potential, but the default behavior of AFL wastes much computation power on frequent paths, resulting in slow and fruitless mutation. AFLFast fixes the problem by assigning more computation power to seeds exercising rare paths. SAFL combines them together and is engineered to automatically build hardened executable, run symbolic execution to produce high-quality initial seeds, then run guided fuzz targeting rare paths to improve efficiency.

III. Testing Targets and Procedures

The test plan of `libmsg` mostly focuses on the protocol handlers by targeting the external networking interface. The interface is chosen as the fuzz testing entrance because of the following four aspects.

- Complexity: the event-driven programming paradigms are closely coupled with low-level networking code. It's expensive to mock the layer and directly feed data to the program.
- Completeness. Protocol handlers are the external interface to the environment, thus all code paths of `libmsg` can be reached from there. Coverage-based fuzzers can find inputs to automatically exercise different paths.
- Convenience. Feeding the program by standard POSIX interface is easy. There's no need to analyze a variety of interface and their conventions.
- Accuracy. Network is an untrusted system boundary. Compared to trusted internal API calls, there are no data transmission or calling conventions. The fact eliminates false positives: any input that results in abnormal system behavior must be vulnerabilities.

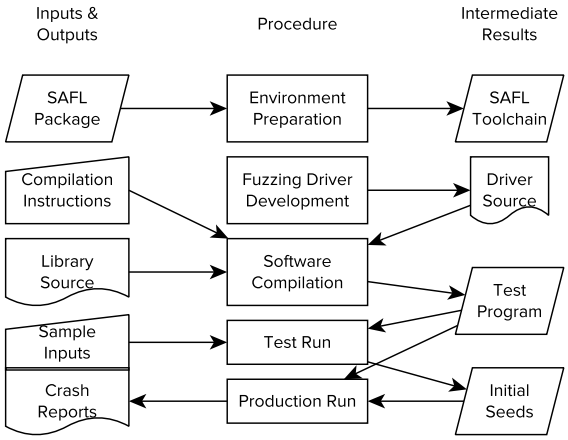

Fig. 1. Stages of test plan

After choosing the testing entrance, the fuzz testing procedure was thought to be easy. As figure 1 illustrates, the first step is preparing the environment and setting up the fuzzers such as SAFL. In the second step of fuzzing driver development, fuzzing driver is written by a test engineer. In the third step of software compilation, source code such as `libmsg` is compiled into a hardened test program for fuzzing. In the fourth step of test run stage, some handwritten inputs are fed to the program to validate the fuzzing driver. The verification is necessary, because a malfunctioning fuzzing driver leads to false positives and false negatives. Symbolic execution is employed to generate high-quality initial seeds in the meantime. Finally, in the production run stage, the improved fuzzer efficiently tests the target program, and produces crash reports to the test engineer.

SAFL tries to automate all the stages as much as possible, therefore test engineers are supposed to run the fuzz testing fairly easily. However, things don't go as planned. From environment preparation to production run, virtually we stumbled on every step. The following section lists the obstacles we encountered and the corresponding solutions.

IV. Typical Obstacles and Solutions

Obstacle 1: Configuration Inconsistency in Environment Preparation Stage. Huawei heavily customizes their operating system in both production and development environment. SAFL are built to run any recent Linux distributions. But in this case, glibc of the operating system in Huawei is too old to run. Similar situations also exist in many other companies such as CRRC Corp. Ltd.

Solution: Rebuild the System from Source. Initially, we try to rebuild the whole project. However, GCC provided by the operating system fail to build LLVM, which is a requisite

for instrumentation framework. Finally, fresh GCC and CMake are built from source, and the problem is resolved.

During the rebuilt process, a powerful machine is helpful. Fuzzing is a computing-intensive job, so is the build procedure of SAFL's supporting tool modules. For example, it takes half an hour just to build LLVM. With adequate software and hardware environment, the following steps are finished more efficiently.

Obstacle 2: Absence of Training in Driver Development Stage. `libmsg` is a messaging library, which means it can only run together with a fuzzing driver. For many test engineers, fuzzing is still a new technique. Although they have domain-specific knowledge, training is needed to develop fuzzing driver. Developing fuzzing driver from scratch is expensive. For example, to test `libmsg`, on the one hand, a sample application should be developed, which initializes the library and registers callbacks to exercise interfaces. On the other hand, a simple request sender should be developed to transport the data generated from the fuzzer to the library.

Solution: Transform Fuzzing Driver from Unit Tests and Sample Code. We find that existing unit tests and sample code snippets are helpful. Not much effort is required converting unit tests focusing on external interfaces to fuzzing drivers. Exploiting sample program takes even less effort. As figure 2 shows, in this case we first copy the example program, then add networking code. Net addition counts less than 10 LoC.

```
// Sample code kept as-is
class MyProcess ... {}

// main function is renamed,
int old_main(int argc, char **argv) { ... }

// ... then splitted into LLVMFuzzerInitialize
// and LLVMFuzzerDeinitialize
void LLVMFuzzerInitialize() { /* setup code */ }
void LLVMFuzzerDeinitialize() { /* cleanup code */ }

// New function: feed data into the library
void LLVMFuzzerTestOneInput(uint8_t *buf, size_t size) {
  int fd = socket(AF_INET, SOCK_STREAM, 0);
  connect(fd, &gSvrAddr, sizeof(gSvrAddr));
  send(fd, buf, size, 0);
  close(fd);
}
```

Fig. 2. Convert sample program to fuzzing driver

Obstacle 3: Build System Complexity in Software Compilation Stage. Different from most open-source projects used in fuzzing benchmarks and tutorials, `libmsg` has a complex build procedure. Due to historical burden, the build scripts are a mix of bash, autoconf and CMake. It's tedious to manually modify different scripts to build hardened executable. This situation is quite common in real industry projects, and reduces the usability and acceptance of fuzzing techniques in practice.

Solution: Develop Automatic Toolchain to Support Complex Compilation and Build. We solve the problem by providing an automated toolchain. The toolchain recognizes different build systems and injects instrumentation-related compiler options following the standard customization interface exposed by the build system. The user-friendly utility hides internal complexities, reducing the knowledge required to operate these tools. It isn't limited to the fuzz engine SAFL, other fuzzers like AFL and AFLFast can also benefit from hardened executable using our toolchain. Table I shows that the usage is pretty close to normal toolchain invocation. Only a prefix is inserted at the beginning of each normal build command.

TABLE I
SAFL AND NORMAL BUILD PROCEDURE

	SAFL	Normal build
Prepare	safl-prepare	
Configure	safl-configure ./configure	./configure
Build lib	safl-build make -j	make -j
Build driver	safl-build g++ target.cc -c	g++ target.cc -c
Link	safl-link target.o lib.a	g++ -o app target.o lib.a

There are two special threats in `libmsg` that would fail the toolchain, which needs to be patched ahead. The first is the unmaintained build script. `libmsg` uses a complex shell script to fetch and configure the depended software, compile the dependencies, run unit tests then build deployable packages. The 600-line shell script is lack of maintenance, which even reports errors on successful build. We separate the core component out, which enables manual build to use SAFL.

The second is antipatterns in CMake. SAFL injects compilation flags to harden the executable. The procedure is automated using the standard compiler flag customization support from various build tools. But the nonstandard use of CMake cripples our tool. If there are advanced configurations not covered by CMake, developers may use the standard interface to add compiler options. However, `libmsg` takes the worst approach that directly overwrites the low-level variable passed to compilers. We replaced the problematic CMake commands to fix this problem.

Obstacle 4: Shallow Bugs in Test Run Stage. When fuzzing driver is built, a test run should be performed. In this case, a few correct and random inputs are made by hand casually, then passed to the program. This step verifies that the fuzzing driver fills data into `libmsg` as expected, and the API conventions are followed. Unfortunately, even the most simple inputs can crash `libmsg`.

Solution: Shallow Bugs Repair with Developers. Vulnerabilities residing in shallow paths must be fixed before the deployment of a fuzzer. If bugs on frequent paths crash the program, deep paths can't be exercised. We collaborate with the developer to fix the shallow bugs to prevent immediate crashing, and bug hunting on deep paths can be continued.

Obstacle 5: Bug-Hiding Code in Production Run Stage. We closely monitor the fuzzer running on production server. To our surprise, we can still find a tricky condition that misguides the fuzzer. `libmsg` uses GLOG, the Google logging module, for logging and aborting on assertion failure. The default behavior is dumping the call stack then exit with status code 1. However, as figure 3 shows, we find a commit that disables the error reporting and directly exit with status code 0. As a messaging library, `libmsg` is expected to handle errors and run continuously until explicitly told to exit. It's a denial of service attack if the external input can cause the program to exit.

```
// Call to print stacktrace is omitted
// Exit with status code 0 instead
void FailureFunction() {
  exit(0);
}

// Called on system initialization
void ProtocolP::init() {
  // ...

  // Replaces default behavior on fatal situation
  google::InstallFailureFunction(FailureFunction);

  // ...
}
```

Fig. 3. Code snippet disabling error reporting

Solution: Patch the Program and the Fuzzer to Detect Hiding Bugs. To enable capturing this type of problem, we first work on the program side, removing the commit that disables the error reporting mechanism. We further work on the fuzzer side, because American Fuzzy Lop — of which SAFL is built on top — only detects critical signals by default. We patched `run_target` in `afl-fuzz` to enable detection of nonzero exit code. AFL based fuzzers can also use this patch to capture abnormal exit on server-side programs.

V. Results

After overcoming the obstacles, we successfully run our fuzzer on the production server. We evaluate the effectiveness of fuzz testing and traditional unit tests on the very same hardened messaging library `libmsg`. One executable is built for the fuzzing test, and another is built with the built-in unit tests.

Table II shows that SAFL successfully detects 11 vulnerabilities, including 9 previously unknown ones, while traditional unit tests only capture 3 vulnerabilities. Generally speaking, without considering the cost and risk to overcome the obstacles, fuzz testing is more effective than traditional unit tests.

TABLE II
Number of vulnerabilities discovered by SAFL and unit test

Type	SAFL	UT
Incorrect exit procedure	2	0
File descriptor leak	1	0
Reachable assertion	1	0
Unaligned memory access	2	2
Uncontrolled memory allocation	4	0
Use after free	1	0
Memory leak	0	1
Total	11	3

One memory leak vulnerability is the only case where fuzzing failed to capture. Unit tests are handwritten, which have the advantage of systematic coverage of interfaces and APIs. On the contrary, our fuzzing driver relies on the sample code. Due to the simplicity of the sample code designed for educational use, some part of the program is bound to be unreachable. We believe that a better fuzzing driver would help solve this problem.

Two Unaligned memory access vulnerabilities are captured by both techniques. Figure 4 shows the code snippet of a vulnerable utility function responsible for reading an integer from network. The protocol parser calls it every time a new connection is established, therefore the buggy code is executed on almost all inputs. It's unsurprising that vulnerabilities residing on frequent paths are easy to capture with both techniques.

```
// data isn't guaranteed to be aligned!
void *data = consume_bytes(4);
uint32_t size = *reinterpret_cast<uint32_t *>(data);
```

Fig. 4. Utility function containing undefined behavior

The other vulnerabilities are only discovered by fuzzing. For example, the file descriptor leak is not detected by unit tests, because unit tests usually feed the program with a small number of predefined inputs. The number of inputs is too small to use up all the file descriptors, and the symptom never shows up. The vulnerability of reachable assertion demonstrates another weakness of unit tests. The root cause behind is a design fault that routes connection of the new protocol to old protocol handlers. Many components are involved in the complex implementation. On the one hand, unit tests don't care about system-level design decisions, but the interaction between various components is error-prone. On the other hand, once design faults occur, it's difficult to defeat cognitive inertia and write test cases for the corner case.

VI. Discussion

Valuable lessons are learned during the practice. Observations on fuzzing preparation include:

1) **Most industrial environments are different and may not satisfy the prerequisites for fuzzers.** Unlike static analyzers working on source code, fuzzing is a dynamic approach, therefore it's inevitable to run fuzzers in the environment where programs are designed to run. The operating system, compiler, and hardware may vastly differ from everyday development environment. We find that advanced techniques used in "smart" fuzzers usually have limitations in industrial environments. For example, the popular symbolic executor KLEE [3] doesn't support multithreading and inline assembly, which greatly limits its industrial use. Intel PIN [9], a popular dynamic instrumentation toolkit, also have restrictions on CPU feature and kernel version. For practical use, there's lots of work ahead.
2) **Support for complex build systems is lack and highly needed.** Large projects are usually accompanied with complex build systems. Instrumented builds require modification in compiler flags. Although most build systems provide standard interfaces to do so, they are unreliable due to antipatterns in build scripts. More low-level approaches may be taken, such as compiler customization.

Observations on writing fuzzing drivers include:

1) **Training is needed to write qualified fuzzing drivers.** To adapt fuzz testing to existing projects, one major hindrance is the absence of fuzzing drivers. Test engineers

or software developers usually have enough domain-specific knowledge, but training on fuzzing is required to convert their knowledge to working fuzzing driver.

2) **Sample code and unit tests help write fuzzing drivers.** Sample code in tutorials covers the core concepts and workflow, and unit tests provide a detailed view of each component. Converting sample code to fuzzing driver is a cost-effective approach to start, while unit tests help to cover more advanced APIs and interfaces not mentioned in sample code. Combine them together, and high-quality fuzzing drivers can be written with a lower cost.

Observations on running fuzzers include:

1) **Code may intentionally hide bugs.** Dirty hacks are ubiquitous in industrial projects. Low-level interventions such as signal handler and callback on fatal logs may help to solve problems at the moment, but the long lasting side effect prevents fuzzers to detect anomalies. These workarounds are hard to locate in source code, but must be found and bypassed.
2) **Resource usage should be monitored.** Crash is a direct evidence for the presence of bugs. However, resource utilization anomaly may indicate bugs as well. For instance, after investigating the exceedingly high number of execution timeouts, we found the uncontrolled memory allocation vulnerability. The kernel overcommits memory when the allocation is too large, which leads to slowdown on access. It's probable that resource usage is another effective metric to guide coverage-based greybox fuzzers.

VII. Conclusion

In this paper, we present an empirical study on adapting fuzzers to real industry projects, which differ greatly from standard projects commonly used by researchers to evaluate the efficiency of fuzzers. The program itself and the environment that the program runs in contain many pitfalls hardly mentioned in any existing studies.

During the phases of environment preparation, driver development, software compilation, test run and production run, we list the typical obstacles and discuss the solutions in detail. After overcoming those obstacles, we run the fuzzer efficiently and report 9 previously-unknown bugs. By analyzing the results, we compare unit testing and fuzz testing, and confirm that fuzzing can be low-cost yet achieve better results than unit tests. The observations and solutions can also be applied to the fuzzing practice of other real industrial projects.

References

[1] *add fuzz target for individual ffmpeg APIs for in-process fuzzing with libFuzzer , AFL, and similar fuzzing engines*. https://lists.ffmpeg.org/pipermail/ffmpeg-cvslog/2016-November/102728.html. [Online; accessed 09-January-2018]. 2016.

[2] 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.

[3] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[4] Sang Kil Cha, Maverick Woo, and David Brumley. "Program-adaptive mutational fuzzing". In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 725–741.

[5] *Fuzzing for Security*. https://blog.chromium.org/2012/04/fuzzing-for-security.html. [Online; accessed 09-January-2018]. 2012.

[6] *How SQLite Is Tested*. https://www.sqlite.org/testing.html. [Online; accessed 09-January-2018]. 2017.

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

[8] *libFuzzer – a library for coverage-guided fuzz testing*. https://llvm.org/docs/LibFuzzer.html. [Online; accessed 07-January-2018]. 2018.

[9] Chi-Keung Luk et al. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Acm sigplan notices*. Vol. 40. 6. ACM. 2005, pp. 190–200.

[10] Kostya Serebryany Mike Aizatsky, Abhishek Arya Oliver Chang, and Meredith Whittaker. *Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software*. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html. [Online; accessed 09-January-2018]. 2016.

[11] Matt Morehouse. *Set of tests for fuzzing engines*. https://github.com/google/fuzzer-test-suite. [Online; accessed 07-January-2018]. 2016.

[12] Yan Shoshitaishvili et al. "(State of) the art of war: Offensive Techniques in Binary Analysis". In: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, pp. 138–157.

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

[14] Mingzhe Wang et al. "SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing". In: *Software Engineering Companion (ICSE-C), 2018 IEEE/ACM 40th International Conference on Software Engineering*. IEEE. 2018.

[15] Michal Zalewski. *American fuzzy lop*. 2015.