國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

實現可再現的模糊測試

Toward Reproducible Fuzzing

謝啟仁

Chi-Jen Hsieh

指導教授: 蕭旭君 博士

Advisor: Hsu-Chun Hsiao Ph.D.

中華民國 111 年 8 月

August, 2022

doi:10.6342/NTU202203969

國立臺灣大學碩士學位論文
口試委員會審定書

實現可再現的模糊測試

Toward Reproducible Fuzzing

本論文係謝啟仁君（學號 R09922022）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 111 年 8 月 4 日承下列考試委員審查通過及口試及格，特此證明

口試委員：
蕭旭君

黃世昆 （指導教授）

黃俊穎

系 主 任 洪士灝

doi:10.6342/NTU202203969

doi:10.6342/NTU202203969

Acknowledgements

特別感謝蕭旭君教授的指導，提供了許多想法與建議，讓我獲益良多。也要感謝網路安全實驗室的同學提供的幫助與鼓勵。

doi:10.6342/NTU202203969

doi:10.6342/NTU202203969

摘要

模糊測試是一種自動化偵測軟體漏洞的技術，許多模糊測試工具已經被開發出來，並且成功地辨識出真實世界軟體中的關鍵漏洞。然而，因為模糊測試工具中的運算邏輯存在著非確定行為 (例如隨機生成的測試輸入與隨時間變化的條件件)，研究人員難以驗證關於模糊測試工具的說法 (例如更好的代碼覆蓋率或是發現更多的漏洞)。目前的研究人員只能透過進行多次重覆實驗，並檢查結果是否一致來驗證相關說法。

為了使驗證的過程更簡單，這篇論文探討了具有可重複性的模糊測試。可重複性指的是經過相同的計算過程並產生完全一致的結果，這使得捏造數據或是偽造結果變得更加困難。這篇論文顯示出我們可以在不影響其功能和性能的情況下使得模糊測試具有可重複性。

為了實現這一目標，我們首先找出了使模糊測試無法重現的因素，並將其分為五類: 隨機性、環境、時間、平行化和目標程式。然後我們對每個因素提出補救措施。按照所提出的準則，我們將 AFL 修改成可重複性的版本並稱之為 *ReAFL*。我們的評估表明，*ReAFL* 成功地重現了各種目標程序的模糊測試實驗。此外，*ReAFL* 在實驗階段和重現階段都取得了與 AFL 相當的性能。這篇論文可做為引導使得其他研究人員可以自行將自己所進行的模糊測試實驗改為具有可重複性的版本。

doi:10.6342/NTU202203969

關鍵字：模糊測試、再現性

doi:10.6342/NTU202203969

Abstract

Fuzzing is a technique to automate the discovery of software vulnerabilities. Many fuzzing tools have been developed and successfully identified critical vulnerabilities in real-world software. However, claims about fuzzing tools are sometimes hard to validate because they have ingrained non-deterministic behaviors in their algorithmic logic, such as randomly generated test inputs and time-dependent conditions. To validate such a claim (e.g., better code coverage or more bugs found), researchers today will repeat the experiment multiple times and see whether the results are consistent.

This work aims to ease this validation process by exploring the concept of reproducible fuzzing. Reproducibility requires generating identical computational procedures and results, making it harder to fabricate data or falsify results.

We show that it is possible to make fuzzers reproducible without affecting their functionality and performance. To achieve this, we first identify factors that make the fuzzing non-reproducible and group them into five categories: randomness, environment, time,

doi:10.6342/NTU202203969

parallelization, and target program. We then propose remediation for each factor. Following the proposed guideline, we modify AFL to support reproducibility, and the resulting tool is called *ReAFL*. Our evaluation shows that *ReAFL* successfully reproduces the fuzzing results on a wide range of target programs. Also, *ReAFL* achieves comparable performance to AFL during both the fuzzing and reproduction phases. Our work can serve as a guideline for developing reproducible fuzzers.

Keywords: Fuzzing, Reproducibility

doi:10.6342/NTU202203969

Contents

doi:10.6342/NTU202203969

doi:10.6342/NTU202203969

doi:10.6342/NTU202203969

doi:10.6342/NTU202203969

List of Figures

doi:10.6342/NTU202203969

doi:10.6342/NTU202203969

List of Tables

doi:10.6342/NTU202203969

doi:10.6342/NTU202203969

Chapter 1 Introduction

Fuzzing is an automated testing technique to find bugs and vulnerabilities in software. It automatically generates a large amount of random data as testcases to test the target program and observes if any bugs or crashes occur. Today, fuzzing is one of the most promising techniques and is used to find bugs in various types of real-world software. As of July 2022, OSS-Fuzz [18] has discovered over 40,500 bugs in 650 open-source projects since its launch in December 2016. Because of its popularity and usability, many researchers are dedicated to making fuzzers more efficient and powerful.

While fuzzing techniques are constantly being improved, the community has struggled to reproduce the fuzzing results, partly due to the randomness of fuzzing. Reproducibility is important for scientific research. It allows other researchers to validate the results and makes the work easier to be analyzed and extended. In the previous work of fuzzing, the authors usually only provide descriptions of the environment and evaluations of the results without the method to reproduce their results. Therefore, other researchers can only use the information in the paper to speculate on the actual running process, or they need to make multiple attempts on their own to achieve similar fuzzing results before analyzing the vulnerabilities. If the probability of a specific outcome is not sufficiently high, it may take more time and resources to produce similar results. For example, suppose the probability of a fuzzer finding a specific bug is ten percent. In that case, researchers

doi:10.6342/NTU202203969

can only encounter it once every ten times and observe how the fuzzer generates the input of this bug. The lack of reproducibility also makes people difficult to find the overclaimed benefits of a paper, leading to wrong or misleading evaluations.

This paper presents a general guideline to make the fuzzing experiment reproducible.

Before going any further, we need to describe the type of reproducibility we want to achieve. We focus primarily on “reproducibility” as defined by the National Academies of Sciences, Engineering, and Medicine [15], that is, “obtaining consistent results using the same input data; computational steps, methods, and code; and conditions of analysis.” In the fuzzing, this means that we can obtain the same results (including code coverage, seed corpus, crashes, etc.) with the same resource budget, initial input seed, target program, and random seed information. To achieve this, we need the whole fuzzing process, including target program feedback, seed mutate, seed scheduling, etc., to be consistent with the fuzzing phase.

For consistency throughout this paper, we introduce two terms, “the fuzzing phase” and “the reproduction phase.” The fuzzing phase refers to the original run of the experiment. The reproduction phase refers to the reproduction run of the experiment. We aim to regenerate the same input sequence as the fuzzing phase using roughly the same resource in the reproduction phase.

Although we can log the details of the fuzzing phase, such as the timeout, execution, and mutation, and reproduce the complete process step by step according to the record, it will require considerable space to log all the details. What is more serious is that using this method to reproduce may not achieve the original performance. We believe that the efficiency of the fuzzer is also one of the results that need to be considered. Therefore, the

doi:10.6342/NTU202203969

ideal approach is to keep the minimum data and maintain the same (or better) efficiency during the reproduction phase.

To improve the reproducibility of fuzzing techniques, we analyzed the fuzzing process and found out why the fuzzing is challenging to reproduce. Then we divide the factors that affect reproducibility into five categories: randomness, environment, time, parallelization, and target program.

Randomness Because fuzzing is inherently random, the outcomes of a fuzzing can be quite different each time. The fuzzer cannot generate the same input sequence without controlling the randomness, which makes it non-reproducible.

Environment For an experiment to be reproducible, the researcher must capture the environment in which the fuzzer was built. However, Klees et al. [10] found that "most papers treated the choice of seed casually, apparently assuming that any seed would work equally well, without providing particulars." The lack of information about the execution environment and configuration makes other researchers hard to reproduce the results.

Time Many fuzzers use the execution time of seeds to determine the strategy and resource allocation. It is hard to reproduce a fuzzing because we cannot guarantee that the program execution time will be the same every run.

Parallelization Running fuzzers in parallel is an excellent way to improve the performance when running on a multi-core system. However, the presence of race conditions makes the fuzzer unable to get the right input seeds from other fuzzers in the reproduction phase, which makes parallel fuzzing non-reproducible.

doi:10.6342/NTU202203969

Target Program All the above four categories will affect the target program too. If we cannot get the same output from the target programs in the reproduction phase, the results of the whole experiment will be different.

These factors will be discussed in detail in the subsequent section.

We propose the corresponding solutions for each factor and present a reproducible fuzzer architecture. For environment and target program, in the fuzzing phase, the fuzzer needs to record sufficient information about the environment, parameters, and settings. And for randomness, the fuzzer needs to ensure the random number generator can provide the same random number set in the reproduction phase. To solve the time problem, we propose "BasicBlockCounter" and "BasicBlockOut" to replace execution time and timeout. Our approach is based on a simple yet intuitive observation: the more basic blocks are executed, the more time the execution takes. Hence, replacing the execution time with the number of basic blocks executed will have a similar effect. For reproducibility, execution time and timeout are challenging to reproduce because even the same input may lead to different results, which are different from basicblock count. For a reproducible target program, the same input will go through the same control flow; That is, the same input will have the same basic block count. In summary, using the number of basic blocks makes it easier to reproduce while maintaining the same effect and strategy compared to the execution time. As for parallelization, we can successfully reimplement the process of different fuzzers exchanging seeds without race conditions occurring in the reproduction phase by recording the necessary information in the fuzzing phase.

To prove the feasibility, we implemented a proof of concept on AFL [20], one of the most popular fuzzers, and called it *ReAFL*. We tested *ReAFL* on three real-world programs

doi:10.6342/NTU202203969

and three Magma benchmark [7] programs and compared them with AFL. The results show that *ReAFL* has reproducibility and can reproduce experiments conducted on five of the six target programs. And the performance of *ReAFL* is similar to that of AFL, which shows that the implementation of our design does not cause a significant decrease in performance. Finally, we also compared the efficiency difference between the fuzzing phase and the reproduction phase. In our experiment, the time spent to reproduce the complete fuzzing phase was at most 5% more than the original time, showing that the reproduction efficiency was not much worse than that of the fuzzing phase.

Our methods are applicable to general fuzzers, and we hope that in the future, more and more fuzzers can become reproducible by referring to our solutions.

In summary, we make the following contributions:

- We analyzed the fuzzing process and identified factors that undermine reproducibility.
- We presented a general guideline to overcome these factors during fuzzer design and implementation.
- Following our guideline, we developed *ReAFL*, a fuzzer supporting methods reproducibility, by modifying AFL, a popular fuzzer.
- We plan to evaluate *ReAFL* on different benchmarks to validate its correctness and evaluate its efficiency.

doi:10.6342/NTU202203969

doi:10.6342/NTU202203969

Chapter 2 Background & Related Work

2.1 Reproducibility

The definition of reproducibility varies in literature and is often confused with replicability. Stanford geophysicist Jon Claerbout defined [4]"reproducing" as "running the same software on the same input data and obtaining the same results." And he also defined "replicating" as "re-implementing a new software based on the method provided in the original publication, and running it with the same input data to obtain similar results." On the other hand, Association for Computing Machinery (ACM) [1] slices these concepts into three kinds: repeatability, reproducibility, and replicability. In this paper, we use the terminology proposed by National Academies of Sciences, Engineering, and Medicine (NASEM), which is the closest to our situation.

According to their definition, reproducibility is the ability to use the same data, input, and other details to repeat the same computational procedures and generate identical results. In the field of fuzzing, computational procedures contain the execution, mutation, seed schedule, feedback analysis, etc., performed during the fuzzing process. At the same time, results refer to the state at the end of the fuzzing, which contains code coverage,

doi:10.6342/NTU202203969

found seeds, and crashes. In addition, fuzzing experiments are all about how fuzzer can produce good results in a comparative time. Therefore, we believe that for the fuzzing, the result also includes efficiency; if the time spent in the reproduction phase is too much than the fuzzing phase, it is not a successful reproduction.

NASEM also define the meanings of replicability. Replicability is the ability to "obtain consistent results across studies aimed at answering the same scientific question, each of which has obtained its own data." For the field of fuzzing, the consistent results refer to achieving similar code coverage or the ability to find same bugs. If we run an experiment several times and the average of the code coverage or triggered bugs are similar to the original experiment results, the experiment has replicability.

In summary, we believe that a reproducible fuzzer must be able to duplicate the code coverage and bug counts in any experiments using sufficient details of environment settings and inputs. And the time spent in the reproduction phase must be as close as possible to the time spent in the fuzzing phase. In this definition, the method of logging all computational processes and replaying them one by one cannot be used because it is difficult to reproduce the computational process in the same efficient way by logging; Also, it requires too much logging. In this paper, our goal is to achieve a reproducible fuzzer while using as little additional record data as possible and maintaining the same efficiency as possible.

2.2 Related work

FuzzBench [14] is a platform for evaluating fuzzer performance. Everyone can easily integrate a fuzzer with FuzzBench and compare it with other fuzzers. It provides real-

doi:10.6342/NTU202203969

world software as benchmark programs and uses both code coverage and bug counts for comparison. Every fuzzer running on FuzzBench has the same computing power by using Google's compute resources. And it claims that it provides reproducible experiments by pinning the versions of both fuzzers and target programs. Different from our work, the reproducibility they claimed refers to replicability. When a user runs an experiment on FuzzBench multiple times, the user can get an average score similar to the original one. But the user may not obtain the same coverage and same seeds for every reproduction run.

AFL++ [5] is a fork of AFL. In addition to the features of AFL, AFL++ added lots of techniques from other state-of-art fuzzers to make it more powerful. AFL++ also claims that it has reproducibility by fuzzing the target with the same random seeds. After testing, we found that it can generate the same testcases at the beginning of the fuzzing, but after a while, it can not reproduce the same testcase sequence because the occurrence of timeouts is uncontrollable.

In other domains, researchers hold reproducibility challenges to encourage people to share their research results that are reproducible. For example, in the field of machine learning, Neural Information Processing Systems (NeurIPS) [17] [16] conference introduced a program to improve reproducibility in machine learning research. The program provided a code submission policy, a reproducibility challenge, and a checklist. These tools help researchers check whether their work provides enough complete information for reproduction. For another example, in the field of information retrieval, European Conference on Information Retrieval (ECIR) [8] encourages researchers to submit papers that repeat, reproduce, generalize, and analyze prior work. It also claims that "reproducibility is key for establishing research to be reliable, referenceable, and extensible for the future."

doi:10.6342/NTU202203969

In fuzzing field, some researchers are beginning to focus on the importance of reproducibility. In 2022, the first International Fuzzing Workshop (FUZZING) [6] introduce preregistration [2], which makes the reproducibility of the submitted work more important. It also suggested that the evaluation of high quality papers must be soundness and reproducible.

doi:10.6342/NTU202203969

Chapter 3 Factors of Non-reproducibility

We identify common factors of non-reproducible fuzzing results and group them into five categories. This section describes each of them in detail. Later in Section 8, we will also discuss other non-reproducibility factors that we are currently unable to handle.

3.1 Randomness

A basic factor of non-reproducibility is randomness. Because many fuzzers rely on random number generators to produce inputs to target programs, select mutation strategies, and schedule input seeds, the input sequence constructed in the reproduction phase tends to be different from fuzzing phase.

There are two types of random number generators, each requiring different handling to ensure reproducibility.

3.1.1 Pseudo-random number generators

One is pseudorandom number generators (PRNGs), whose output sequence is determined solely by an initial value or seed. In other words, by using the same seed, one can

doi:10.6342/NTU202203969

easily re-generate the same sequence of numbers.

3.1.2 True random number generators

The other is true random number generators (TRNGs), which extract randomness from external sources such as timings of keystrokes or thermal noise. Because of their non-determinism, their output sequence is considered true random. Unlike PRNGs, there is no efficient way to re-generate the output of TRNGs.

For example, AFL and AFL++ use a PRNG provided by the C library to obtain random numbers. After generating 50000 to 100000 numbers, they will switch to a TRNG provided by Linux to create a new random seed. The mixed-use of the two further complicates the reproduction of the fuzzing results.

3.2 Environment

A fuzzer may exhibit different behaviors when running in different environments, including fuzzer configurations, software dependencies, and low-level systems.

3.2.1 Configuration

A fuzzer often uses parameters and environment variables to configure customized features or strategies. The details of the initial seed corpus are also vital for the fuzzing. Using different initial seed corpus for the same target program will lead to quite different performance [10]. Thus, using the same parameters, environment variables, and initial seed corpus is vital for generating consistent results in the reproduction phase.

doi:10.6342/NTU202203969

3.2.2 Software Dependency

If a fuzzer uses additional packages or plug-ins, their versions and settings must remain the same. For example, AFL++ has a feature called NeverZero. In complex programs, the counters that collect the edge coverage may be zero at the end of the program because it was filled up and wrapped around, which causes the fuzzer to think that the edge has not been traveled mistakenly. NeverZero will add an extra 1 when a counter is wrapped around so that the path traveled is never zero. If the user's LLVM [11] version is 8.0 or lower, AFL++ will not use NeverZero due to the performance costs. However, if the user's LLVM version is 9.0 or greater, AFL++ enables NeverZero because the implementation will be optimal in LLVM 9.0 or greater. Therefore, the different versions of LLVM will make AFL++ use different features, and other strategies, which affects the performance and results.

3.2.3 Low-level System

Some fuzzers invoke functions that depend on low-level systems, such as OSes, filesystems, and hardware. Thus, researchers may encounter non-reproducibility when conducting the reproduction phase on different low-level systems.

For example, AFL uses the readdir functions to read the other fuzzing instances' directories when running in the parallel mode. However, the readdir system call does not specify the order of the returned data. When using readdir to read a folder, the order of the returned data depends on the underlying filesystem. Thus, if readdir returns a different order of seeds in the reproduction phase, the fuzzing result will also differ.

doi:10.6342/NTU202203969

3.3 Time

As time is a scarce resource in fuzzing, many fuzzing strategies are time-dependent. For example, an input that runs for more than a timeout threshold (e.g., 1 second) may be deemed wasteful and get discarded.

As we will show later, time is the most challenging factor of non-reproducibility in the reproduction phase because attempting to control it may affect the core logic of fuzzing.

3.3.1 Execution Time Per Input

Fuzzers often use an input's execution time to assess the input's quality and take actions accordingly. For example, both libFuzzer [13] and AFL prioritize seeds with shorter execution times and allocate more resources to them to improve efficiency. Thus, the variance in an input's execution time will affect the resource allocation and seed scheduling and result in non-reproducible fuzzing results.

Also, fuzzers usually discard inputs whose execution time is longer than a timeout to avoid being blocked or hanged. If timeout events occur inconsistently in the fuzzing and reproduction phases, the results will diverge. More subtlely, when a timeout occurs during AFL's trimming operation (which aims to shorten a seed while maintaining code coverage), AFL will use the original seed without trimming. Inconsistent timeout events, in this case, will result in different seed lengths and possibly affect the seeds generated subsequently.

doi:10.6342/NTU202203969

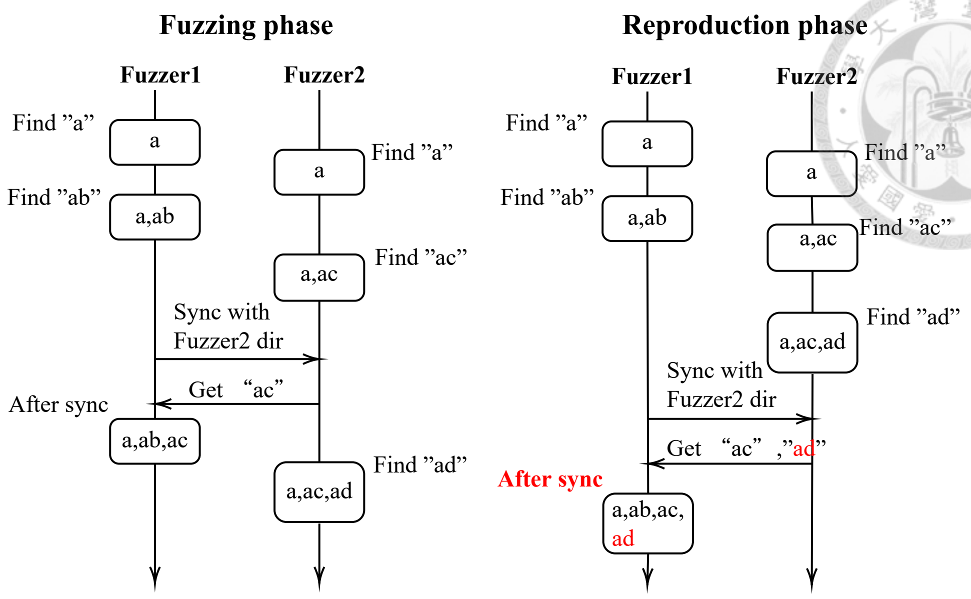


Figure 3.1: A race condition example when reproducing parallel fuzzing

3.3.2 Overall Execution Time

Besides an input's execution time, some fuzzers use the overall execution time to determine when to adjust their fuzzing strategies. For example, AFL will adjust its mutation strategy after running for ten minutes, and AFL++ will switch to a different mutation strategy if no new coverage is found for a long time. Because there is no guarantee that a fuzzer will always make the same progress (e.g., generating the same number of seeds) within the same time budget, such time-based conditions will lead to non-reproducible results.

3.4 Parallelization

Parallelization improves resource utilization and computing performance in a multicore system. Some single-core fuzzers, such as AFL and libFuzzer, provide a parallel mode for running multiple instances simultaneously on multicore systems. Several recent fuzzers, such as PAFL [12] and EnFuzz [3], support better synchronization among paral-

doi:10.6342/NTU202203969

lelized instances, each of which can use a different strategy or explore a different region of code. We deem parallel fuzzing reproducible if each parallelized instance can generate the same input sequence in both the original and reproduction phases.

In Parallel fuzzing, fuzzer instances will exchange their produced seeds with each other to share progress. There are two common approaches to sharing seeds between instances. One uses the same corpus folder, and the other keeps separate corpus folders per instance but regularly synchronizes them. Both may become non-reproducible due to race conditions among fuzzer instances. Figure 3.1 shows an example: Fuzzer1 obtains a seed from Fuzzer2 in the synchronization round during the fuzzing phase, but Fuzzer2 does not generate that seed before the end of the synchronization round during the reproduction phase. This leads to diverged results.

3.5 Target Program

The aforementioned factors may also affect the target program's reproducibility. If the target program behaves inconsistently between the original and reproduction phases, the fuzzing results may be inconsistent.

Addressing those aforementioned non-reproducibility factors will also help improve the target program's consistency. However, as we will see in the next section, some require modifying the program logic, which may not be an option if such modification affects the vulnerabilities we intend to find. For example, some fuzzers aim to find race conditions or time bombs in the target program. In this case, the fuzzing results will be questionable if we modify how the target program handles multi-processing or time-based conditions. Consequently, we put "target program" into a separate category.

doi:10.6342/NTU202203969

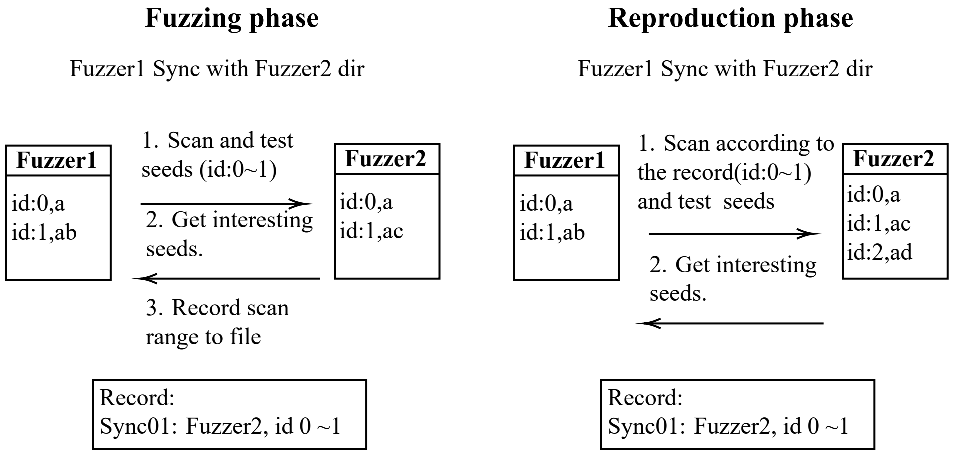
Chapter 4 Design Guidelines

This section presents design guidelines to address the non-reproducibility factors listed in Section 3.

4.1 Randomness

4.1.1 PRNG

To eliminate the inconsistency introduced by PRNGs, one can record every random seed used in fuzzing phase and reuse it in reproduction phase. AFL++ has implemented an option to support this method.



Fuzzing phase
Fuzzer1 Sync with Fuzzer2 dir
Fuzzer1
id:0,a
id:1,ab
1. Scan and test seeds (id:0~1)
2. Get interesting seeds.
3. Record scan range to file
Fuzzer2
id:0,a
id:1,ac
Record:
Sync01: Fuzzer2, id 0 ~1
Reproduction phase
Fuzzer1 Sync with Fuzzer2 dir
Fuzzer1
id:0,a
id:1,ab
1. Scan according to the record(id:0~1) and test seeds
2. Get interesting seeds.
Fuzzer2
id:0,a
id:1,ac
id:2,ad
Record:
Sync01: Fuzzer2, id 0 ~1

Figure 4.1: The record of synchronization avoids the happen of race conditions.

doi:10.6342/NTU202203969

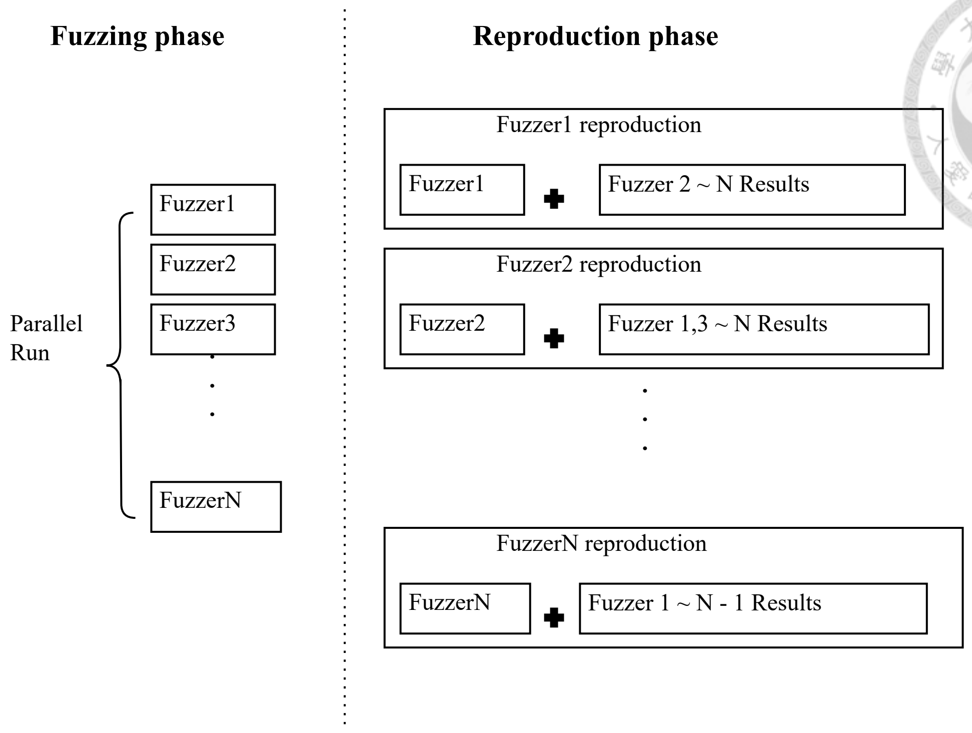


Figure 4.2: Each fuzzer will run with the finished results of other fuzzers to reproduce.

4.1.2 TRNG

A TRNG outputs true random bits that cannot be regenerated from a random seed. Thus, one needs to save the TRNG's output in fuzzing phase and directly reuses the saved output in the reproduction phase.

4.2 Environment

We recommend recording the entire environment used in the fuzzing phase, including parameters, variables, software dependency, and low-level systems. The easiest way is to use virtualization tools such as Docker. Research can build the container image for others to produce the same environment as the fuzzing phase. We also recommend integrating the commands of reproduction into an automation script. Automation scripts can make sure that the user uses the same parameters as in the fuzzing phase.

doi:10.6342/NTU202203969

4.3 Time

To deal with non-reproducibility caused by time, we recommend replacing wall-clock time with measures closely related to fuzzer operations. More concretely, we will explain using the number of **executed basic blocks** to measure the duration of target program execution; and using the **total number of execution** to measure the duration of the fuzzing procedure itself.

4.3.1 Execution Time Per Input

We introduce a basic block counter, BBCounter, which measures the number of executed basic blocks. We use BBCounters to replace the use of wall-clock timers. This replacement is reasonable in most scenarios because the execution time increases with the number of executed basic blocks. We will later discuss scenarios (e.g., infinite loops within one basic block) where BBCounter may be unsuitable.

Unlike a wall-clock timer, the number of basic blocks that a target program executes for a given input is always the same (assuming that the target program itself is reproducible). Therefore, fuzzers that require evaluating inputs by their execution duration (for example, used for seed scheduling or resource allocation) can now perform this evaluation consistently in both the fuzzing and reproduction phases.

Similarly, we introduce a basic block timeout, BBOut, to replace a timeout. Thus, if a fuzzer wants to discard inputs that take too long to execute, it can abort the program after executing more than a threshold number (i.e., the BBOut value) of basic blocks. Unlike the timeout set by a wall-clock timer, the trigger condition of BBOut is consistent and easy

doi:10.6342/NTU202203969

to reproduce.

It is also possible to use instructions as the measurement unit. We use basic blocks instead of instructions to reduce instrumentation overhead, as shown in the implementation section. Some optimizations speed up fuzzing by reducing the number of basic blocks that need to be instrumented [9]. However, since we need to track the program's progress in real-time, omitting the insertion of basic blocks would make it impossible to count the number of passed basic blocks. Therefore, we cannot use this kind of optimization to reduce our overhead.

Now, we discuss the case that a program hangs inside a basic block, such as waiting for I/O. We will be unable to detect this type of hangs by using BBOut. To solve this problem, we need to use a **hang timeout** to detect hang events, but we treat hang events with special care to ensure reproducibility. Specifically, during the fuzzing phase, if a hang occurs and is caught by the hang timeout, the fuzzer records the ID of the current execution. During the reproduction phase, if the ID of the current execution is marked as a hang execution, the fuzzer skips this run and reports directly as a hang. If the current execution is not marked, the fuzzer gives it a ten times hang timeout to execute because the previous results showed that this input should not be stuck in a basic block. A ten times hang timeout may still be triggered and may lead to non-reproducible results, but at least it can prevent the fuzzer from getting stuck and completing the whole experiment.

Although this solution is less than ideal, we believe it has little effect on the soundness of reproducibility because hang events do not occur frequently. In our evaluation, not a single hang timeout had occurred.

doi:10.6342/NTU202203969

4.3.2 Overall Execution Time

Although BBCounter can approximate how long a target program has been running for a given input, it does not account for the overall execution time of the fuzzing, which includes the time taken by the fuzzer itself and the accumulated time for testing multiple inputs.

To make them reproducible, we should replace the overall execution time with the total number of execution. There are three advantages of using total number of execution. First, this method can achieve the same effect when the correct number is set. Second, this approach is easier to adjust for different targets to achieve better results. Lastly, this method is also easy to reproduce.

4.4 Parallelization

In parallel fuzzing, the fuzzing and reproduction phases may yield inconsistent results due to race conditions among parallel-running fuzzer instances.

To ensure consistency, we must satisfy two conditions: The first point is that the timing of the synchronization is consistent. For example, libFuzzer uses time as an indicator of whether sync is required or not. In this case, we must use the design proposed in Section 4.3 to make the synchronization timing the same, whether in the reproduction phase or the fuzzing phase.

Second, we must ensure that the result of each synchronization in the reproduction phase is the same as in the fuzzing phase. To achieve this goal, we must first understand the fuzzer synchronization process.

doi:10.6342/NTU202203969

When the fuzzer synchronizes, it scans the target seeds of this synchronization and selects the ones it wants (such as those that can increase coverage) to add them to its own seed pool. We can reproduce the synchronization process by recording the scanned range during synchronization. The details are as follows:

In the fuzzing phase, each fuzzer instance must record the range of seeds scanned in each synchronization. When it comes to the reproduction phase, each fuzzer instance can use its own record to reproduce the same synchronization process. As shown in Figure 4.1, in the fuzzing phase, fuzzer1 recorded the range of seeds scanned in the first synchronization and scanned the same content in the reproduction phase according to the record.

As mentioned above, our method can guarantee that the range of seeds in each synchronization is correct, but if the seeds are not read-only, it means that the content of the seeds synchronized in the reproduction phase may not match the content of the fuzzing phase, which will fail to reproduce. Therefore, we must ensure that each seed file is read-only. Our proposed method is very simple; when the fuzzer needs to change the content of the seed, we save the modified seed to a new file instead of overwriting it. In this way, the fuzzer will not change the original content of the seed and make the seed read-only.

In addition to preventing race conditions, efficiency is also essential. However, the reproduction phase may run much slower than the fuzzing phase due to such locking behavior in the solutions mentioned above. To speed up reproduction, we suggest reproducing each fuzzer instance independently by emulating its exchange with others. As shown in Figure 4.2, each fuzzer records its own synchronization details during the fuzzing phase. Regarding the reproduction phase, each fuzzer will run with the results of other fuzzers.

doi:10.6342/NTU202203969

For example, fuzzer1 will use its record plus the complete results of fuzzer2 to N to reproduce the process of fuzzer1.

4.5 Target Program

Similar to recording the fuzzing environment, we need to record the version and the parameter of the target program. However, suppose the target program is non-reproducible. In that case, we cannot guarantee that entering the same input into the target program will lead to the same result, which makes fuzzing challenging to reproduce. One method is to modify the code of the target program to make it reproducible. The other is to make reproducibility one of the necessary conditions when selecting a target program. Target programs that use time, multi-thread or TRNG cannot be reproduced. For example, OpenSSL cannot be reproduced because it uses TRNG to create random numbers for encryption.

doi:10.6342/NTU202203969

doi:10.6342/NTU202203969

Chapter 5 Implementation

To demonstrate the feasibility of our proposed design, we implement a fuzzer called *ReAFL* to support reproducibility. This section presents our implementation based on AFL with LLVM-based instrumentation.

We have modified 1775 lines of AFL source code. The main modified parts are: random seed related, synchronization related, seed corpus score related, and timeout related source code. We also changed the instrumentation inserted in the compile phase for BBOut and BBCounter. In the process of modifying, we mainly look for keywords or functions related to the factor mentioned in Section 3 and check whether they need to be changed; for example, search for words like "timeout, random seed function" or "synchronize" in the source code. If the fuzzer still cannot reproduce the correct result after being changed, we then trace the code and experiment to find out the reason that affects reproducibility.

5.1 Randomness

AFL uses random numbers during both compilation and fuzzing. We will explain how to record random numbers and regenerate them during the reproduction phase.

When compiling the target program, AFL uses a PRNG provided by the C library to

doi:10.6342/NTU202203969

generate a basic block ID and decide whether to instrument a basic block if AFL_INST_RATIO is set. For the latter, AFL_INST_RATIO is a parameter that indicates the ratio of instrumented basic blocks. Setting a small ratio will reduce the instrumentation overhead and thus speed up fuzzing.

We add the functions of recording and reading random seeds to the *ReAFL* compiler, so that we obtain identical instrumented binaries after compilation.

During fuzzing, AFL uses both PRNG and TRNG. It uses a TRNG to create new random seeds and a PRNG to generate random numbers from the seeds. Although we can record all the random numbers generated by TRNG and used in the reproduction phase, this will reduce efficiency. Therefore, we refer to the practice of AFL++. When AFL++ is in “Fix Seed mode,” its PRNG will use a single random seed from beginning to end. Therefore, it is not necessary to use TRNG to create more random seeds for PRNG. In this way, *ReAFL* only needs to record a random seed to reproduce the same set of numbers successfully.

$$\underbrace{sync:\ 2}_{\textbf{Sync Times}}\ ,\ \overbrace{Fuzzer\ 2}^{\textbf{Target Fuzzer}},\ \underbrace{7\ \sim\ 10}_{\textbf{Seeds file range}};$$

Figure 5.1: The record of synchronization.

5.2 Parallelization

For synchronization timing, AFL uses cycles to judge whether synchronization is necessary. When a fuzzer has used all seeds in the queue, it is called a cycle. The cycle is independent of time, so it can ensure that the synchronization timing of the fuzzer is fixed.

doi:10.6342/NTU202203969

When AFL is in the parallel mode, each fuzzer will have an independent folder to save its own seeds. Fuzzer will read the folders of other fuzzers to get the seeds when it is synchronized. In addition, AFL will name seeds in the order in which they are produced when saving seeds.

To prevent race conditions, *ReAFL* will record the details of the synchronization and the range of the seeds when synchronizing. As shown in Figure 5.1, in the second synchronization, fuzzer1 reads the 7th to 10th seeds of fuzzer2; *ReAFL* will record it as "sync:2 fuzzer2 id 7-10" and use it as a basis to reproduce the fuzzing in the reproduction phase.

Saving the record is not enough to reproduce the results due to the trimming phase. We must ensure that the seeds are read-only. AFL will trim the input seed to make it smaller and more efficient after saving it, which will change the content of the input seed file. Unlike AFL, *ReAFL* will trim the input seed before saving it. Since each input seed will only be trimmed once, changing the order of trim execution will not cause different results and will make it read-only.

Finally, to avoid dragging down the performance by other fuzzer, *ReAFL* will run with the finished results of other fuzzers to reproduce. We provide an automated script that will reproduce each fuzzer instance using the record of each fuzzer instance and the results of other fuzzer instances to implement the methods mentioned in Section 4.4.

5.3 Environment

We provide docker images to reproduce the fuzzing. The docker image will prepare the environment, the fuzzer, and the target program. To easily reproduce the results of the fuzzing, we provide an automation script that records information such as the parameters

doi:10.6342/NTU202203969

executed in the fuzzing phase.

AFL uses the function readdir when scanning and reading folders, which relies on low-level systems. However, readdir does not guarantee the reading order; the order depends on the implementation by the underlying filesystem. In *ReAFL*, we replace readdir with scandir, which guarantees reading in the alphabetical order.

5.4 Time

5.4.1 Execution Time Per Input

To eliminate the timing factor, we implement `BBCounter` and `BBOut` in *ReAFL*. `BBCounter` keeps track of the number of basic blocks executed, and `BBOut` defines the maximum number of basic blocks that the target program can execute before being killed.

To support both, we insert several lines of instrumentation code at the beginning of each basic block of the target program. The instrumentation code will increment the BBCounter and check whether BBOut has been reached. If so, the program will terminate. We use an LLVM pass [11] to insert the instrumentation code, shown in Listing 1.

```
// Enter the basic block
$Counter \leftarrow Counter + 1$;
if $Counter > BasicBlockOut$ then
    exit();
end
```

Listing 1: Basic block Counter

Besides adding instrumentation, we also modify AFL's code to remove reliance on wall clocks.

doi:10.6342/NTU202203969

Execution time AFL calculates the score of an input seed based on its execution time. A seed with a higher score (shorter execution time) will obtain more resources. In this spirit, *ReAFL* replaces the execution time in the calculation with the number of basic blocks passed, obtained from BBCounter.

Timeouts *ReAFL* uses `BBOut` to replace the original timeout. *ReAFL* will calculate the ratio of time to basic blocks and convert timeout to `BBOut` in the fuzzing phase. When reproducing, *ReAFL* will automatically set `BBOut` to the same value as the fuzzing phase, ensuring that computers with different computer power can get the same results. *ReAFL* still uses a large timeout in the fuzzing phase to prevent hanging within a basic block. If the *ReAFL* timeout expires in the fuzzing phase, it records the current execution time. In the reproduction phase, *ReAFL* checks whether the execution was timeout or not before running it. *ReAFL* skips the execution and reports it as timeout if the execution was marked by the record. Otherwise, *ReAFL* runs it without setting a timeout.

5.4.2 Overall Execution Time

AFL switches to a different havoc mutation method after running for ten minutes. Since ten minutes is very short for a fuzzing experiment and will not affect efficiency, *ReAFL* directly removes this flag according to the practice of AFL++.

5.5 Target Program

The docker image will provide information such as the parameters and the version of the target program. For those non-reproducible targets, we will try to use LD_PRELOAD

doi:10.6342/NTU202203969

to overwrite the original non-reproducible library functions. For example, mktemp is used to create a temporary file or directory and will use the current time to produce random file names. In the reproduction phase, we have no control over the time factor, so we cannot create the same file name. Different file names will cause the target program to go through different basic blocks, which will lead the reproduction to fail. We rewrite a reproducible version of mktemp to use PRNG instead of the current time to generate it, so we can generate the same filename by setting the same random seed. And we use LD_PRELOAD to overwrite the original function. In this way, when the target program uses mktemp, it will be changed to use our changed version.

If it is not because of the library function that cannot be reproduced, we will also try to change the code directly to make the target program reproducible. For example, Magma [7] is a ground-truth benchmark in which the Target program has not only lots of real bugs but also inserted instrumentation to enable people to check whether the inputs have triggered bugs. Magma provides a tool to collect the number of times each bug has been passed or triggered when fuzzing the target program. This tool allows us to regularly check the progress of the fuzzer and make it easier to compare with other fuzzers. However, this tool will make the currently running target program copy the number of times each bug was passed or triggered to shared memory when passing the instrumentation section, further causing target programs to have different control flows and affecting reproducibility. To prevent it from affecting reproducibility, we have to change the instrumentation code in the target program. Originally, the instrumentation would only copy the data to shared memory at regular intervals. After our changes, the data is copied to shared memory every time, which makes the control flow of the target program consistent and can be reproduced correctly.

doi:10.6342/NTU202203969

Lastly, if the above two methods are complicated to make the target program reproducible, we do not use them as the testing target.

doi:10.6342/NTU202203969

doi:10.6342/NTU202203969

Chapter 6 Evaluation

We performed an evaluation on *ReAFL* to examine its ability to reproduce the results and the impact of design on its performance. In particular, we aim to answer the following research questions:

RQ1 Can *ReAFL* reproduce the results?

RQ2 Will the modification of AFL significantly affect its performance?

RQ3 Will the execution time of the fuzzing phase significantly differ from the reproduction phase?

RQ4 What is the size of the extra records used for the reproduction phase?

6.1 Environment

Experiments were performed on Ubuntu Server 18.04 LTS with an AMD Ryzen 9 5950X Processor (16 cores) and 128 GB of RAM.

doi:10.6342/NTU202203969

6.2 Target Selection

We selected target programs from the Magma benchmark and open source programs in the real world to test our implementation. Magma is a ground-truth fuzzing benchmark that collects the history bugs from the target and inserts them into the same version. We modified the instrumentation of the magma benchmark programs so that the control flow of the target program is not affected when we use the tool to get information about the bugs. The detailed implementation is in Chapter 5. We are using our modified version to ensure its reproducibility.

6.3 Experiment

We performed *ReAFL* and AFL with LLVM-based instrumentation 20 times for each target selected previously. We refer to each of these runs as *trials* and each trial runs 24 hours by default. In addition to the standard mode, we examined the parallel mode of *ReAFL* and AFL with one master instance and one secondary instance. *ReAFL* and AFL in parallel mode were also run 20 times on real-world targets. Finally, we reproduced every trial run by *ReAFL*.

Since AFL++ provides a fixed seed mode and claims that it has reproducibility, we also use AFL++ to perform 20 runs against the real-world programs and reproduce each run.

We plot the code coverage over time, and the interval shows a 95% confidence interval around the mean coverage. We use the edge coverage calculated by AFL to evaluate the code coverage.

doi:10.6342/NTU202203969

Targets	**Seed Files**	**Crashed Files**	**Code Coverage**
readelf	283214/283214	0/0	1310720/1310720
objdump	89359/89359	0/0	1310720/1310720
nm	72310/72310	0/0	1310720/1310720
readelf-parallel	1273190/1273190	0/0	1310720/1310720
objdump-parallel	259477/259477	0/0	1310720/1310720
nm-parallel	215666/215666	0/0	1310720/1310720
libpng	29521/29521	1307/1307	1310720/1310720
libtiff	50698/50698	1879/1879	1310720/1310720
libsndfile	15564/26686	272/532	1300920/1310720

Table 6.1: ReAFL results of reproduction phase on real world programs and magma benchmark.

Targets	**Seed Files**	**Crashed Files**	**Code Coverage**
readelf	68543/389404	0/0	512244/563200
objdump	21852/120995	0/0	1049631/1076480
nm	4851/117036	0/46	667858/709120
readelf-parallel	44273/878270	0/0	1067283/1126400
objdump-parallel	34984/307706	0/0	2103616/2152960
nm-parallel	11590/353535	0/88	1354058/1418240

Table 6.2: AFL++ results of reproduction phase on real world programs.

RQ1: Reproduciblity To ensure reproducibility, we used *ReAFL* to reproduce each trial and compared the results of fuzzing phase and reproduction phase, including crashes, seeds, and code coverage (bitmap). To be regarded as successfully reproduced, the results of the reproduction phase must be identical to the fuzzing phase.

Table 6.1 shows that eight out of nine (except libsndfile) the results of reproduction phase are the same as those of fuzzing phase; Column seed files represent the number of successfully reproduced seeds and the number of seeds generated in the fuzzing phase; Column crashes files represent the number of successfully reproduced crashes files and the number of crashes files generated in the fuzzing phase. Since *ReAFL* did not find any crashes on real-world programs, so it was considered a successful reproduction. Column code coverage represent the number of successfully reproduced bitmap bytes and the total number of bitmap bytes. For AFL, the bitmap size is fixed to 65536 bytes, so the total

doi:10.6342/NTU202203969

bitmap bytes is 65536 multiplied by 20 times. We can see that all the experiments except libsndfile have the same code coverage in the reproduction phase as in the fuzzing phase. From the above, we can judge that *ReAFL* can correctly reproduce the experiment in both normal and parallel modes. Although we only use edge coverage as the basis for determining code coverage, we can actually reproduce any kind of coverage since we reproduced the entire fuzzing phase process and the results.

We found that libsndfile uses time-dependent functions, which causes it to go through a different number of basic blocks even if we feed it the same input, making our reproduction fail.

And from Table 6.2, we can see that AFL++ can only reproduce the first few minutes of the fuzzing phase and will fail to reproduce in the middle because of the time factor, so it cannot reproduce successfully in seed files, crashes files, and code coverage.

	Coverage			**Executions per Second**		
Targets	AFL	ReAFL	Decrease	AFL	ReAFL	Decrease
readelf	19.63%	19.35%	1.41%	3992.6	3769.2	5.6%
objdump	11.38%	11.58%	-1.74%	1998.4	2167.8	-8.48%
nm	10.47%	10.05%	4.08%	1573.2	1494.5	5.00%
Average			1.25%			0.71%

Parallel	**Coverage**			**Executions per Second**		
Targets	AFL	ReAFL	Decrease	AFL	ReAFL	Decrease
readelf	20.30%	20.42%	-0.59%	2866.5	3239.2	-13%
objdump	12.16%	12.34%	-1.51%	1599.9	2207	-37.95%
nm	10.87%	10.90%	-0.35%	1240.9	1566	-26.19%
Average			-0.82%			-25.56%

Table 6.3: Comparison of *ReAFL* and AFL on real world programs.

RQ2: Performance Difference of *ReAFL* and AFL To compare the performance of the fuzzers, we evaluated the performance of the fuzzer against the target programs with different evaluation metrics. For Magma, we evaluated using the total bugs found in the

doi:10.6342/NTU202203969

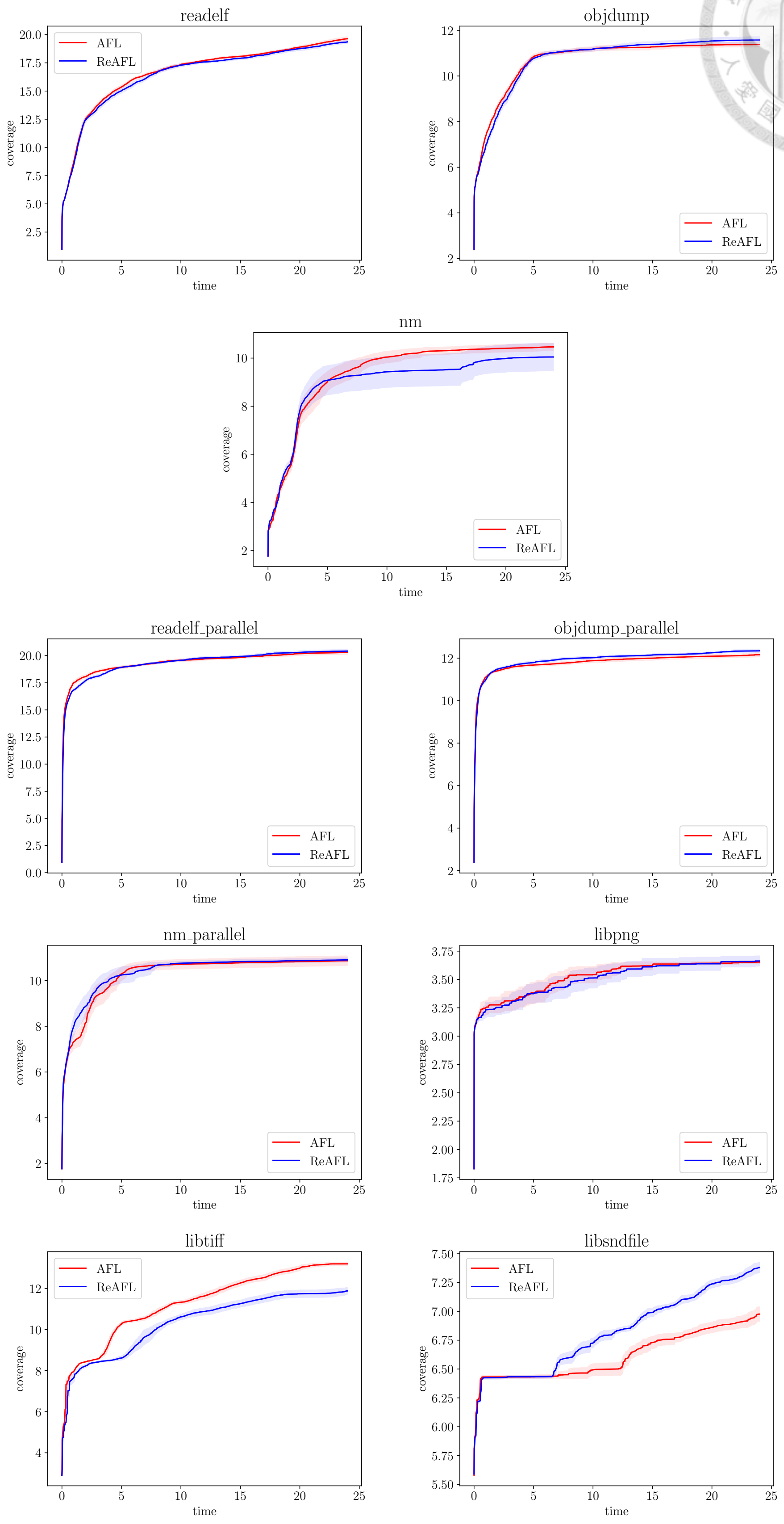


Figure 6.1: Code coverage on real-world program and magma benchmark.

doi:10.6342/NTU202203969

	Coverage			Executions per Second		
Targets	AFL	ReAFL	Decrease	AFL	ReAFL	Decrease
libpng	3.65%	3.66%	**-0.27%**	15026.4	18970.1	-26.24%
libtiff	13.20%	11.88%	10%	1987.4	1153.5	38.31%
libsndfile	6.98%	7.38%	-5.73%	744.3	1221.6	-64.12%
Average			1.33%			-16.13%

	Crashes		
Targets	AFL	ReAFL	Decrease
libpng	59.9	65.35	-9.1%
libtiff	152.3	93.95	38.31%
libsndfile	17.7	26.6	-50.28%
Average			-7.02%

Table 6.4: Comparison of *ReAFL* and AFL on magma benchmark programs.

trials. As for target programs from real-world open-source programs, we use the original code coverage provided by AFL.

Table 6.3 show the difference in performance between AFL and *ReAFL* in real-world programs. From the perspective of Execution per Second, *ReAFL* inserts additional instrumentation into the target to implement `BBCounter`, so the speed of one execution should be slower than that of *ReAFL*. According to Table 6.3, the difference between *ReAFL* and AFL is not greater than 10%. In parallel mode, *ReAFL* even outperforms AFL. From the perspective of code coverage, *ReAFL* is very similar to AFL. We speculate that it is because *ReAFL* uses the same strategy as AFL, and it can be seen from Figure 6.1 that the code coverage of these three targets grows very slowly after 12 hours, so the impact of the difference in the number of executions is limited.

Table 6.4 show the difference in performance between AFL and *ReAFL* in magma benchmark. Since the target programs in the magma benchmark are larger than the real-world programs we selected, the shortcoming that the *ReAFL* basic block counter cannot be accurately converted to time is magnified. Table 6.4 and Figure 6.1 shows that *ReAFL*'s

doi:10.6342/NTU202203969

performance is much worse than AFL on libtiff but much better than AFL on libsndfile. We speculate that this is because the initial basic block out was incorrectly estimated, resulting in a different performance than AFL with a timeout, possibly ending the execution early or spending too much time on a single execution. On average, *ReAFL* is comparable to AFL in terms of coverage, execution per sec, and crashes. We believe that if we can estimate the relationship between basic block and time more accurately, we can simulate the effect of timeout more accurately. In the subsequent Section 7, we will propose a solution to the inability to convert the timeout accurately.

Single Core			
Targets	Fuzzing	Reproduction	Ratio
readelf	24h	24.71h	102.95%
objdump	24h	23.39h	97.45%
nm	24h	24.15h	100.61%
libpng	24h	24.44h	101.86%
libtiff	24h	23.58h	98.25%
Average			100.22%

Parallel Fuzzing			
Targets	Fuzzing	Reproduction	Ratio
readelf	24h	24.23h	100.97%
objdump	24h	24.81h	103.37%
nm	24h	23.52h	97.99%
Average			100.78%

Table 6.5: Comparison of fuzzing phase and reproduction phase on real-world programs and magma benchmark.

RQ3: Difference in efficiency of fuzzing phase and reproduction phase To evaluate its efficiency, we compared the time it took to reproduce the entire experiment with the time spent by fuzzing phase.

Table 6.5 shows that the average time spent in the reproduction phase is within 5% increase of fuzzing phase, which is not much difference.

doi:10.6342/NTU202203969

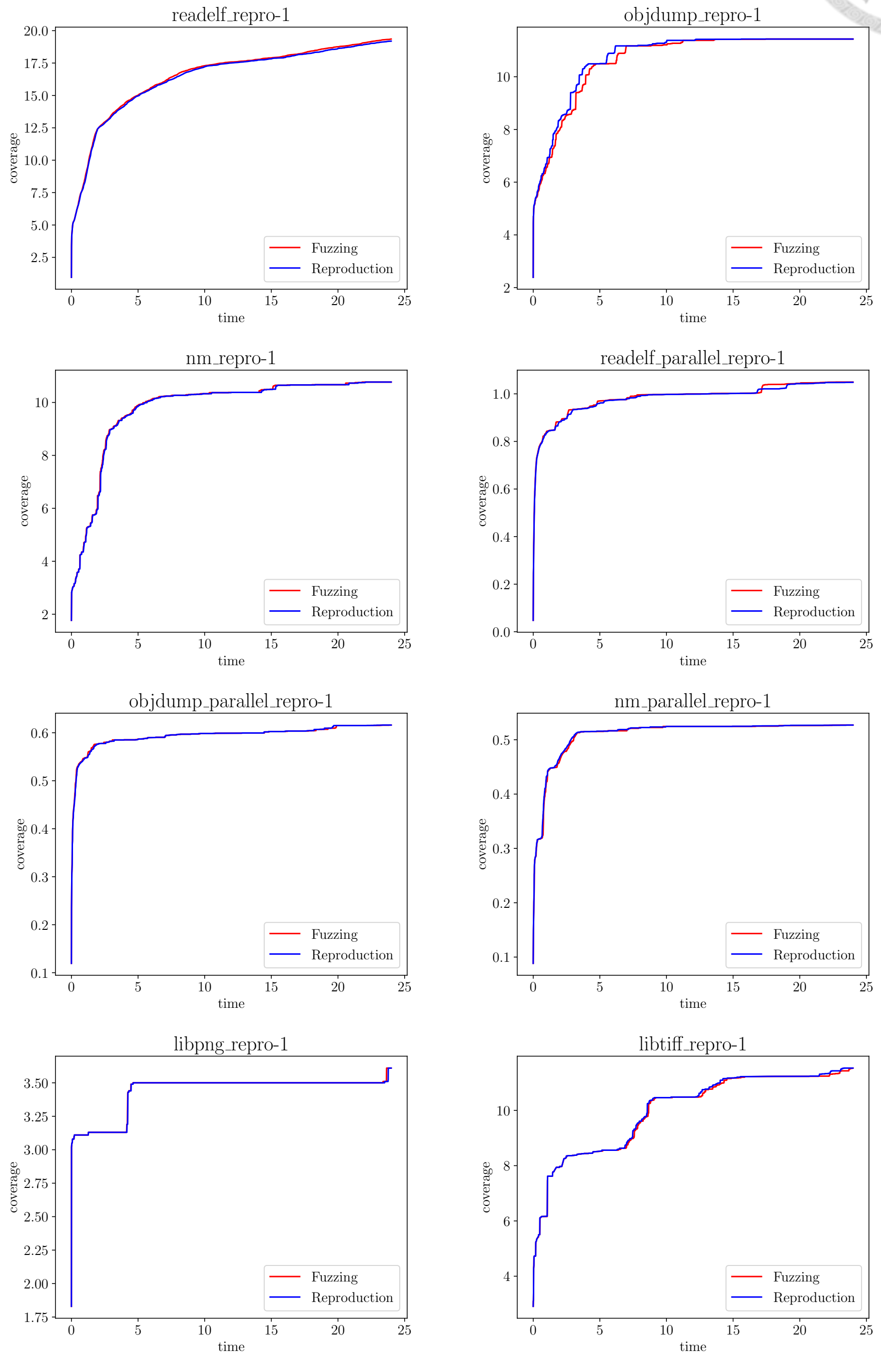


Figure 6.2: Comparison of fuzzing phase and reproduction phase on real-world program and magma benchmark.(For each target program, we randomly selected one of the 20 results to represent.)

doi:10.6342/NTU202203969

To understand whether the process of the reproduction phase is similar to that of the fuzzing phase, for each fuzzing experiment, we randomly selected one of the 20 sets of results to observe. Figure 6.2 shows that most of the code coverages of the fuzzing phase and the reproduction phase are similar at the same time, which means that *ReAFL* also tries to restore the part of the fuzzing phase process in the reproduction phase.

Targets	normal	parallel (master + slave)
readelf	28 Bytes	204.1KB
objdump	27.95 Bytes	126.54KB
nm	27.8 Bytes	62.66KB
libpng	28.8 Bytes	
libtiff	30.45 Bytes	
libsndfile	27.8 Bytes	
Average	28.47 Bytes	131.1KB

Table 6.6: Extra record sizes.

RQ4: What is the size of the extra records used for the reproduction phase? To successfully reproduce the fuzzing experiment, we use the extra records collected in the fuzzing phase during the reproduction phase. The extra record includes random seed, bbout size, and so on. Table 6.6 shows that in the normal mode case, we use a very small amount of extra records for reproduction. But in the parallel mode case, the file is slightly larger because of the synchronization information. In addition, we will reproduce each fuzzer instance independently in parallel mode and need to use the seed files created by other fuzzer instances. Although this will make the amount of extra data spent go up significantly, it is a trade-off to speed up the reproduction phase.

doi:10.6342/NTU202203969

doi:10.6342/NTU202203969

Chapter 7 Discussion

The effect of replacing execution time with BBCounter. The above experiment shows that, for different targets, *ReAFL* will have different performances. For example, when using one core, *ReAFL* performs better than AFL in objdump and libsndfile, while in nm and libtiff, *ReAFL* performs worse than AFL. It is because that *ReAFL* is based on the number of basic blocks passed instead of time, but the time spent on basic blocks varies according to the content of the basic blocks, so it does not simulate the timeout result perfectly. To simulate the timeout more accurately, in future work, we consider not only the number of basic blocks but also the weight of different kinds of instruction. By giving different weights to different types of instruction, we can estimate the approximate time that different basic blocks will take, which is more accurate than just basic blocks. How to set the BBOut that matches the timeout is also very important. For example, AFL is judged by the execution time of the initial seeds on the target program to set how long the timeout is. To calculate the corresponding BBOut, *ReAFL* calculates the ratio of time and number of basic blocks by using the execution time and the number of basic blocks passed of the initial seeds in the target program. And using the ratio to convert the timeout into BBOut. This method relies heavily on the information provided by the initial seeds to calculate the BBOut. In some cases, the BBOut may be too large or too small, making it unable to simulate the timeout accurately. We may be able to adjust the BBOut dynamically (e.g.,

doi:10.6342/NTU202203969

according to the proportion of BBOut triggered, etc.) to make the BBOut performance match the expected timeout.

The effect of our approach on the original fuzzer's ability. The above experiment also shows that our approach does not affect the fuzzing strategy much. All the factors except time rely on recording additional information to make the reproduction successful. For time, since we use BBOUT and BBCounter to replace timeout and execution time, as long as we can accurately estimate the relationship between time and basicblock, we can successfully replace the time without affecting the ability of the fuzzer.

Additional time spent in reproduction phase. We also found that the time spent in reproduction phase is usually more than that spent in fuzzing phase because fuzzing in reproduction phase needs to check the fuzzing end conditions and check if there is a hang frequently. And for parallel fuzzing, reproduction phase adjusts the range of files to be read based on the logs. These additional things cause reproduction phase not perfectly to reproduce the fuzzer speed. Suppose the goal of a fuzzer is to speed up the computational procedures of fuzzing. For example, RIFF [19] can speed up the fuzzer operation while maintaining the same strategy, so that an experiment that originally took 24 hours to run can be completed in only 6.53 hours. In that case, it is difficult to reproduce it using our method because there is no guarantee that reproduction phase can reproduce the speed of fuzzing phase.

Reduce the overhead of making fuzzer reproducible. Following our approach to making fuzzer reproducible still requires a lot of manual changes. Automating more parts of the process (e.g., automatically converting time to the number of basic blocks) or modu-

doi:10.6342/NTU202203969

larizing specific steps would make it easier and more desirable for people to make their fuzzer reproducible. The biggest challenge of automation is how to modify the fuzzer source code; because each fuzzer implementation is different, we can hardly find the code corresponding to the factor. One solution is to list the functions or keywords related to the factor (for example, “srand”, “random” which affects randomness and “time”, “execution time”) to assist in finding the factor. We also considered making AFL++ reproducible in the future work. Since AFL++ is a modular framework, any researcher can add their own features to AFL++ and test them. If we can make AFL++ reproducible, then other researchers can check whether their added features have affected the reproducibility, which reduces the overhead of implementation.

Compare the importance of different factors. Each factor is equally essential for our guide because the whole fuzzing experiment will fail to reproduce if any one of them is not handled properly. Moreover, our definition of reproducibility is only success or failure, meaning there is no indication of the degree of reproducibility. However, some factors only need to be handled in special targets, or fuzzer, such as parallelization factors only need to be handled when using parallel fuzzing. In contrast, the time environment and random number are problems that every fuzzer needs to deal with.

Support more factors. Currently, our guide does not support making fuzzer or target programs that use multithread, TRNG reproducible. We may be able to avoid race conditions during the reproduction phase by recording the order of threads that use the same resource during the fuzzing phase. Although we have proposed a solution in the guide, our method is not as efficient if we need to make a lot of use of TRNG to generate random numbers. To improve efficiency, we can replace the TRNG used in the target program and

doi:10.6342/NTU202203969

fuzzer with the PRNG so that fuzzing can be reproduced by using random seeds during the reproduction phase. In addition, as shown in RQ1 6.3, the current reproducible fuzzer is challenging to reproduce for target programs that use the time factor. We may be able to replace the return value of the time-related function with a fixed value to ensure that the target program can have the same control flow when receiving the same input.

Reduce the time and resources spent on reproduction phase. Fuzzing experiments usually run over a long period of time. If the process of reproduction can be accelerated, it will make it easier and more willing for the researcher to verify the work of others. Currently, we can only completely reproduce an experiment with the same time and computational resources. By cutting a fuzzing experiment into fractions, we can reproduce and check the results from a particular time to a particular time. In that case, we can do a sample check and save time by not having to run the whole experiment. To implement this method, we have to solve some problems, such as how to access the fuzzing state halfway through the run, how to compare the two fuzzing states to see if they are the same, and how to continue fuzzing from the intermediate fuzzing state.

Practical uses of reproducible fuzzing If reproducible fuzzing becomes widespread, it can provide other usages. For example, a conference or workshop can provide a random seed set to prevent researchers from picking a favorable result report when running fuzzing experiments. Alternatively, the researcher can also specify which random seed set they will use when registering reports to increase the persuasiveness of the work.

doi:10.6342/NTU202203969

Chapter 8 Conclusion & Future Work

Due to factors such as time-dependent behaviors and random numbers, fuzzing has always been challenging to reproduce as a complete experiment. In this paper, we list the factors that cause non-reproducibility in the fuzzing and propose corresponding solutions for each factor. Finally, we tested the implementation on AFL and called it *ReAFL*.

The evaluation results show that our method can successfully give reproducibility to AFL on most of the target programs and that *ReAFL* does not lose much performance compared to AFL. The evaluation also indicates that *ReAFL* can reproduce in both parallel and normal modes.

Our proposed guideline for reproducible fuzzing has some future work that can be improved. We will improve our guide to support more types of fuzzer and targets and try to add automation to lower the overhead of implementation, which makes more people willing and able to make their fuzzer reproducible through our work. We will also work toward accelerating the speed of the reproduction phase and explore other use cases of reproducible fuzzing. This way, we can make researchers more willing to value and utilize the reproducibility of fuzzing.

doi:10.6342/NTU202203969

doi:10.6342/NTU202203969

References

[1] ACM. Artifact review and badging - current, https://www.acm.org/publications/policies/artifact-review-and-badging-current.

[2] M. Böhme, □. Szekere, B. Ray, and C. Cadar. Journal special issue on fuzzing:what about preregistration?, Apr 2021.

[3] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su. EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In 28th USENIX Security Symposium (USENIX Security 19), pages 1967–1983, Santa Clara, CA, Aug. 2019. USENIX Association.

[4] J. F. Claerbout and M. Karrenbach. Electronic documents give reproducible research a new meaning: 62nd ann. In SEG Technical Program Expanded Abstracts 1992, 1992.

[5] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. Afl++: Combining incremental steps of fuzzing research. In 14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20), 2020.

[6] fuzzingworkshop. Fuzzing workshop 2022. https://fuzzingworkshop.github.io/.

doi:10.6342/NTU202203969

[7] A. Hazimeh, A. Herrera, and M. Payer. Magma. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 4(3):1–29, nov 2020.

[8] D. Hiemstra, M.-F. Moens, J. Mothe, R. Perego, M. Potthast, and F. Sebastiani. Advances in Information Retrieval: 43rd European Conference on IR Research, ECIR 2021, Virtual Event, March 28–April 1, 2021, Proceedings, Part I, volume 12656. Springer Nature, 2021.

[9] C.-C. Hsu, C. Wu, H.-C. Hsiao, and S.-K. Huang. Instrim: Lightweight instrumentation for coverage-guided fuzzing. 2018.

[10] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.

[11] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In International Symposium on Code Generation and Optimization, 2004. CGO 2004., pages 75–86. IEEE, 2004.

[12] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun. Pafl: Extend fuzzing optimizations of single mode to industrial parallel mode. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, page 809–814, New York, NY, USA, 2018. Association for Computing Machinery.

[13] llvm-admin team. Libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/libfuzzer.html, Aug 2022.

doi:10.6342/NTU202203969

[14] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 1393–1403, 2021.

[15] National Academies of Sciences, Engineering, and Medicine. Reproducibility and Replicability in Science. The National Academies Press, Washington, DC, 2019.

[16] NeurIPS. Reproducibility report for ml reproducibility challenge 2022. https://openreview.net/forum?id=s9ilqhz7hak.

[17] J. Pineau, P. Vincent-Lamarre, K. Sinha, V. Larivière, A. Beygelzimer, F. d' Alché Buc, E. Fox, and H. Larochelle. Improving reproducibility in machine learning research: a report from the neurips 2019 reproducibility program. Journal of Machine Learning Research, 22, 2021.

[18] K. Serebryany. Oss-fuzz-google's continuous fuzzing service for open source software. In 26th USENIX Security Symposium, 2017.

[19] M. Wang, J. Liang, C. Zhou, Y. Jiang, R. Wang, C. Sun, and J. Sun. RIFF: Reduced instruction footprint for Coverage-Guided fuzzing. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 147–159. USENIX Association, July 2021.

[20] M. Zalewski. American fuzzy lop (2.52b). https://lcamtuf.coredump.cx/afl/, Jun 2020.

doi:10.6342/NTU202203969