Reproduce
“POSTER: AFL-based Fuzzing for Java with Kelinci[1]”

Weicheng Wu
wulearn.cs09@nycu.edu.tw
National Yang Ming Chiao Tung University
Hsinchu, Taiwan

Abstract

Fuzzing is a random testing technique that can be used to automatically find the input that crashed the program. AFL[4] is the famous Fuzzer in Grey-box fuzzing, which can be used to make the exploration process more efficient by using code coverage. However, it is only a Fuzzer for C programs, so we wanted to investigate how other programs could make use of the AFL-based implementation.

So we reproduced this paper—Kelinci[1], which is a tool that instrumented to Java programs,and use C to interface. So that no modifications to AFL itself are required. We used the last version of AFL for our implementation and were able to reproduce the bug in the image processing library Apache Commons Imaging.

Keywords: Fuzzing, GreyBox, AFL, Java

1 Introduction

Although Java-developed programs may be less likely to have particularly serious vulnerabilities, such as *ABR* (Beyond Array Bounds Read), *ABW* (Beyond Array Bounds Read) issues in C, there may be Arbitrary Memory R&W vulnerabilities.

However, in Java, if any of these problems occur, at best only an IndexOutOfBoundsException may be triggered, and the difference in severity is very, very high.However, it is not impossible to have a dangerous vulnerability in Java, but the probability is usually not as high, but this does not mean that Java is not worthy of Fuzz.Example:Memory Leak (useless variable keeps taking up Memory), RuntimeException (compiler won’t check for the exception), Deserialization Vulnerability...etc.

1.1 Motivation

We were motivated by the fact that we had practiced AFL on the homework and had a general understanding of its operation, so we wanted to explore further how it could essentially change its operation on other programs.

1.2 AFL Workflow

AFL is the famous grey-box Fuzzer, which is somewhere between a white box and a black box. The white box requires analysis of the entire source code, and therefore the time cost is very high and inefficient; the black box is randomly given to the input without knowing the source code, and then observe the result by the output, so do not know the changes that occur in it, and then test for no purpose.

Algorithm 1 The Overall Algorithm of AFL Workflow

```
1:  procedure FUZZING(Prog, Seeds)
2:      Queue ← Seeds
3:      while true do                                  ▷ Start a loop
4:          for seed in Queue do
5:              if ¬isWorthFuzzing(seed) then
6:                  Continue
7:              end if
8:              for i in 0 to Length(seed) do
9:                  newinput ← MUTATED(seed, i)
10:                 runAndSave(Prog, newinput, Queue)
11:             end for
12:             score ← PERFORMANCESCORE(Prog, seed)
13:             for i in 0 to score do
14:                 newinput ← MUHAVOC(Prog, seed)
15:                 runAndSave(Prog, newinput, Queue)
16:             end for
17:         end for
18:     end while
19: end procedure
20: procedure RUNANDSAVE(Prog, input, Queue)
21:     runResults ← RUN(prog, input)
22:     if newCoverage(runResults) then
23:         addToQueue(input, Queue)
24:     end if
25: end procedure
```

The above algorithm is the general process of AFL. AFL is achieved by using instrument to get the coverage, using bitmap and shared memory. Because of the computation, there are path collision and therefore the coverage is not the actual value.

The step-by-step process of AFL operation is as follows:

Step 1: Instrument from source code compilation to record Code Coverage

Step 2: Select some input files and add them to the input queue as the initial test set

Step 3: "Mutation" of the files in the queue according to a certain strategy
(Bitflip, Arithmetic, Interest, Dictionary, Havoc, Splice)

[1]Kelinci github : https://github.com/isstac/kelinci

Step 4: After mutation, if the file updates the coverage, it is added to the queue

Step 5: The above process will continue in a loop, during which the file that triggered the crash will be logged

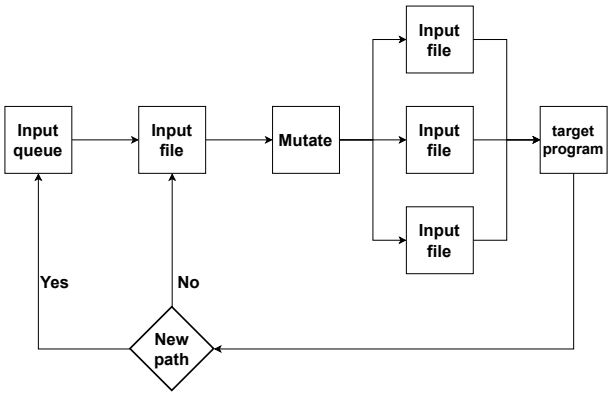

Figure 1. AFL Architecture

2 Overview

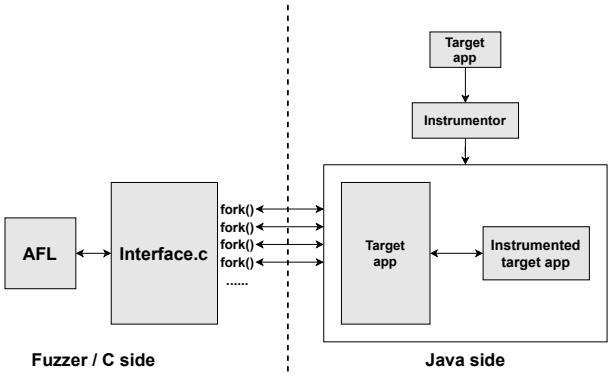

Figure 2. Overview of Kelinci

2.1 Instrumentor

It is an AFL-style tool that implements the AFL instrument approach to Java program, also done with Exclusive-OR calculations and the concept of bitmaps. It can be used to calculate the coverage of a java program, and therefore has the same path collision problem, so the coverage is not the actual value.

2.2 Interface.c

It is a bridge to the target of getting AFL up and running, and it communicates with the Java side of the Driver, which is a server that uses instrumentation to set inputs using parameters and opens a port for future parallelisation.

3 Evaluation

We run on Linux 20.04 with Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz 6-Core Processor machine, and use the final version of AFL for implementation. We run on a single core. The target is Apache Commons Imaging, version release candidate 7 ("commons-imaging-1.0-RC7"), and the goal is to find NegativeArraySizeException[2].

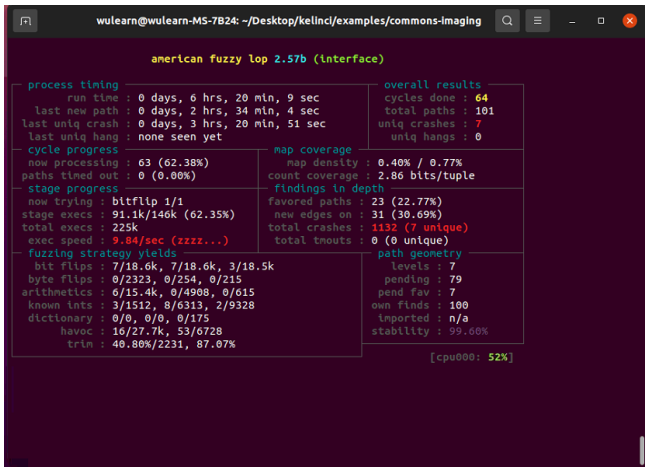

Figure 3. Snapshot of the AFL. The interface is the communication bridge that the Apache Commons Imaging after it has been instrumented and connected to the Driver. We can see that it has been running for six hours and has found seven uniq crashes in three hours.

We have analysed these crashes and basically they fall into two categories: Error reading image & NegativeArraySizeException(Figure 4.), the former of which is a misjudgement by AFL itself. The latter is the result we expected.

Finally, we verified Apache Commons Imaging, version 1.0-alpha1, in the same way to see if the bug was fixed. Figure 5 is the result of actual implementation. We can see that the output of the same input has been corrected to "Error reading image". The bug has been proven to be fixed.

4 Related Work

JQF[2] is a feedback-directed fuzz testing platform for Java. JQF uses the abstraction of property-based testing, which makes it nice to write fuzz drivers as parameteric JUnit test methods. However, the tool is mainly used in the Test phase. It was published in 19, two years after Kelinci published it in 17.

Jazzer[3] is a coverage-guided, in-process fuzzer for the JVM platform developed by Code Intelligence. It is based on libFuzzer and brings many of its instrumentation-powered mutation features to the JVM. It found five CVEs in 2021, but the year of publication was 2021, almost four years after the

[2]The bug report : https://issues.apache.org/jira/browse/IMAGING-203

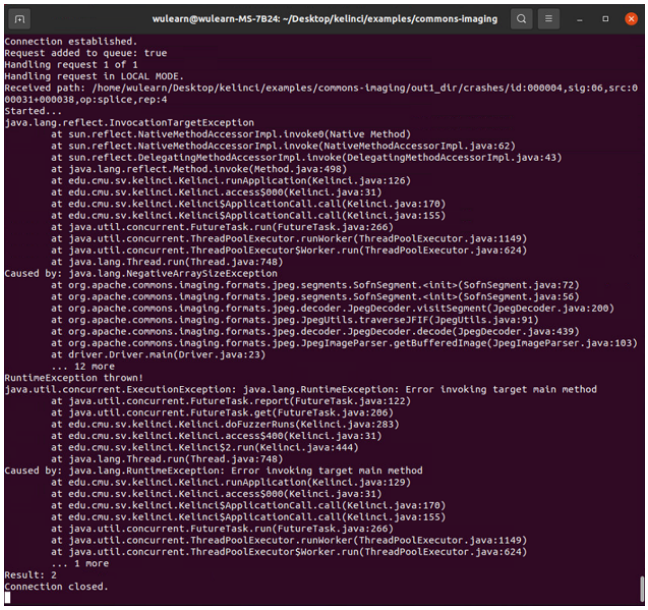

Figure 4. Snapshot of the reason for crashes

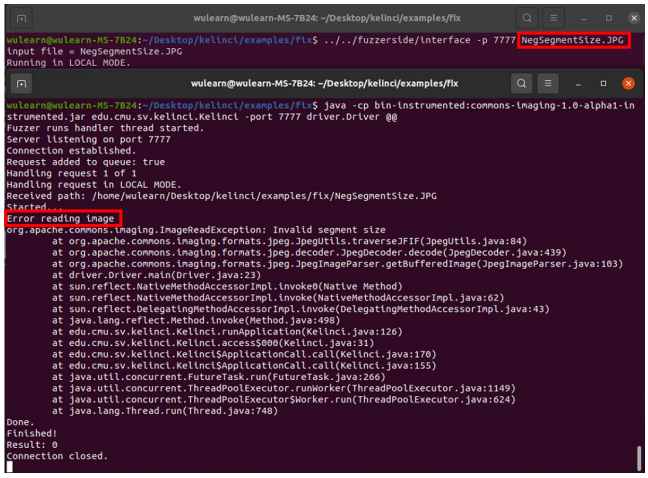

Figure 5. Snapshot of the version 1.0-alpha1

tool was published. Previously, Jazzer also followed Kelinci's approach to Instrumentation.

5 Discussion

Although it is base AFL-style fuzzer, it adds overhead because of the extra TCP protocol and the file I/O, and therefore runs much slower, but still finds bugs in Java programs.

6 Conclusion

We clearly understand the entire Kelinci process and we have successfully reproduced the experimental results of this paper, and have additionally verified the fix for the bug in Apache Commons Imaging, version 1.0-alpha1.

References

[1] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2511–2513.

[2] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 398–401.

[3] Fabian Meumertzheim Sebastian Pöplau Mohammed Qasem Simon Resch Henrik Schnor Khaled Yakdan Sergej Dechand, Christian Hartlage. 2021. Jazzer. https://github.com/CodeIntelligenceTesting/jazzer.

[4] Michal Zalewski. 2017. American fuzzy lop (AFL). *URL: http://lcamtuf. coredump. cx/afl* (2017).