POSTER: AFL-based Fuzzing for Java with Kelinci*

Rody Kersten
Carnegie Mellon University Silicon Valley
Moffett Field, California
rody.kersten@sv.cmu.edu

Kasper Luckow
Carnegie Mellon University Silicon Valley
Moffett Field, California
kasper.luckow@sv.cmu.edu

Corina S. Păsăreanu
Carnegie Mellon University Silicon Valley
NASA Ames Research Center
Moffett Field, California
corina.pasareanu@sv.cmu.edu

ABSTRACT

Grey-box fuzzing is a random testing technique that has been shown to be effective at finding security vulnerabilities in software. The technique leverages program instrumentation to gather information about the program with the goal of increasing the code coverage during fuzzing, which makes gray-box fuzzers extremely efficient vulnerability detection tools. One such tool is AFL, a grey-box fuzzer for C programs that has been used successfully to find security vulnerabilities and other critical defects in countless software products. We present Kelinci, a tool that interfaces AFL with instrumented Java programs. The tool does not require modifications to AFL and is easily parallelizable. Applying AFL-type fuzzing to Java programs opens up the possibility of testing Java based applications using this powerful technique. We show the effectiveness of Kelinci by applying it on the image processing library Apache Commons Imaging, in which it identified a bug within one hour.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Dynamic analysis*; • **Security and privacy → Software and application security**;

KEYWORDS

AFL, Fuzzing, Random Testing, Java

ACM Reference Format:
Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of CCS '17, Dallas, TX, USA, October 30-November 3, 2017,* 3 pages.
https://doi.org/10.1145/3133956.3138820

1 INTRODUCTION

Fuzz testing [6, 10] is an automated testing technique that is used to discover security vulnerabilities and other bugs in software. In its simplest, black-box, form, a program is run on randomly generated or mutated inputs, in search of cases where the program crashes or hangs. More advanced, white-box, fuzzing techniques leverage program analysis to systematically increase code coverage during fuzzing [2, 5, 9]. Grey-box fuzzers leverage program instrumentation rather than program analysis to gather information about the program paths exercised by the inputs to increase coverage. An extensive list of tools can be found here: https://github.com/secfigo/Awesome-Fuzzing#tools. Popular tools available for Java include EvoSuite [3] and Randoop [7].

1.1 AFL

American Fuzzy Lop (AFL) is a security-oriented grey-box fuzzer that employs compile-time instrumentation and genetic algorithms to automatically discover test cases that trigger new internal states in C programs, improving the functional coverage for the fuzzed code [11]. AFL has been used to find notable vulnerabilities and other interesting bugs in many applications and has helped make countless non-security improvements to core tools. For example, AFL was instrumental in finding several of the Stagefright vulnerabilities in Android, the Shellshock related vulnerabilities CVE-2014-6277 and CVE-2014-6278, Denial-of-Service vulnerabilities in BIND (CVE-2015-5722 and CVE-2015-5477), as well as numerous bugs in (security-critical) applications and libraries such as OpenSSL, OpenSSH, GnuTLS, GnuPG, PHP, Apache, IJG jpeg, libjpeg-turbo and many more.

It supports programs written in C, C++, or Objective C, compiled with either GCC or Clang. On Linux, the optional QEMU mode allows black-box binaries to be fuzzed, too. There are variants and derivatives of AFL that allow fuzzing of Python, Go, Rust, OCaml, etc.—yet there is no support for Java programs. The only effort that we are aware of uses the GNU Compiler for Java (GCJ), a free compiler for the Java programming language, that was meant to compile Java source code to machine code for a number of CPU architectures. However, it is no longer maintained [4]. In this paper, we present Kelinci[1], which addresses this gap. We demonstrate the effectiveness of the tool by applying it to the image processing library, Apache Commons Imaging, where it found a bug within one hour of analysis.

2 KELINCI

Inspired by the success of AFL we developed Kelinci, with the goal of applying AFL-style fuzzing to Java based applications. Kelinci provides an interface to execute AFL on Java programs. It adds AFL-style instrumentation to Java programs and communicates results back to a simple C program that interfaces with the AFL fuzzer.

*This material is based on research sponsored by DARPA under agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

[1]Kelinci means rabbit in Indonesian, the language spoken on the Java island.

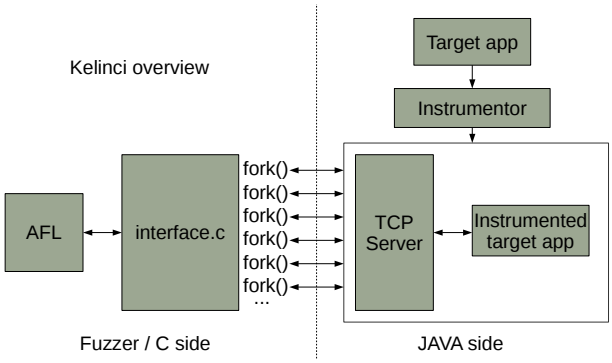

Figure 1: Overview of the design of Kelinci.

It does not require any modifications to AFL, but instead behaves exactly as a C program that was instrumented by one of AFL's compilers. The tool is available at https://github.com/isstac/kelinci.

The overall design of Kelinci is depicted in Fig. 1. The first step when applying the tool is to add AFL-style instrumentation to a Java program. AFL uses a 64 kB region of shared memory for communication with the target application. Each basic block is instrumented with code that increments a location in the shared memory bitmap corresponding to the branch made into this basic block.

The Java version of this instrumentation is the following:

```
Mem.mem[id^Mem.prev_location]++;
Mem.prev_location = id >> 1;
```

In this example, the `Mem` class is the Java representation of the shared memory and also holds the (shifted) `id` of the last program location. The `id` of a basic block is a compile-time random integer, where $0 \leq id < 65536$ (the size of the shared memory bitmap). The idea is that each jump from a block $id1$ to a block $id2$ is represented by a location in the bitmap $id1 \oplus id2$. While obviously there may be multiple jumps mapping to the same bitmap location, or even multiple basic blocks which have the same `id`, such loss of precision is considered rare enough to be an acceptable trade-off for efficiency. The reason that the `id` of the previous location is shifted is that, otherwise, it would be impossible to distinguish a jump $id1 \rightarrow id2$ from a jump $id2 \rightarrow id1$. Also, tight loops would all map to the location 0, as $id \oplus id = 0$ for any id.

Instrumentation is added to the program using the ASM bytecode manipulation framework [1]. Basic blocks are instrumented as described, the `Mem` class is added, as well as a TCP server component, which handles communication with the C side and execution of the target application on incoming files.

AFL expects two pieces of information from an instrumented application. First, the application should be running a fork server, which responds to requests from the fuzzer to fork the process and run the forked process on the provided input file. Second, the application is expected to connect to the shared memory and write to locations corresponding to branches in the program. The `interface.c` component implements a fork server that is identical to the one in programs instrumented by an AFL compiler. When a request to fork comes in, it creates a fork of itself that sends the provided file over to the Java side, receives the result (shared memory plus error status) once the run is done, writes results to shared memory and crashes if the Java program resulted in an exception escaping the `main` method.

For a given input file generated by the fuzzer, the interaction between the C and Java sides is as follows:

(1) `interface.c` receives a fork request from AFL.
(2) One of the forked `interface.c` processes loads the provided input file, the other keeps running the fork server.
(3) The former `interface.c` process sends the provided input file over a TCP connection to the Kelinci server.
(4) The Kelinci server receives the incoming request and enqueues it.
(5) The Kelinci server processes the request by writing the incoming input file to disk, starting a new thread in which the `main` method of the target application is called on the provided input and monitoring it. If the thread throws an exception that escapes `main`, it is considered a bug. If the thread does not terminate within a given time-out, it is considered a hang.
(6) The Kelinci server communicates the results back to the C side. The shared memory bitmap is sent over the TCP connection, as well as the status (`OK`, `ERROR` or `TIMEOUT`).
(7) On the C side, the received bitmap is written to shared memory. Depending on the received status, the program exits normally, aborts or keeps looping until AFL hits its time-out.

To use the tool, the user first runs the `Instrumentor` on the target application. Next, the user starts the Kelinci server in the instrumented program. Finally, the user executes AFL on `interface.c`. The fuzzer is unaware that it is actually analyzing a Java program.

3 EVALUATION

To evaluate Kelinci we ran it on a JPEG parser. We chose Apache Commons Imaging because it is written completely in Java and the Apache Commons libraries are well-known. The version we analyzed is release candidate 7 ("commons-imaging-1.0-RC7"). We compare this against a run of AFL on the djpeg utility that comes with the IJG jpeg library[12].

The first notable observation is that the behavior of Kelinci on Apache Commons Imaging is similar to the behavior of AFL on djpeg: it quickly finds that if the first byte is 0xFF, new behavior is triggered. Using this second-generation test-case, Kelinci finds after approximately 20 minutes that if the next byte is 0xD8, many new behaviors are triggered. In fact, as AFL on djpeg also showed, the two bytes 0xFF 0xD8 that Kelinci finds are the correct markers for the start of a JPEG image. As inputs that cause previously unexplored program behaviors are prioritized by AFL, the fuzzer will get incrementally closer to valid JPEG inputs.

After running Kelinci on our case study for 32 minutes, it found a bug: the value of segment length bytes is not properly validated. Each JPEG segment starts with a two-byte unsigned integer specifying the segment size. As the specified size includes these two bytes, the program subtracts 2 from the size before it is used. It then attempts to allocate a buffer for the segment, which fails (with a `NegativeArraySizeException`) if the specified size is 0 or 1. If this exception is not properly caught by a (server) application using

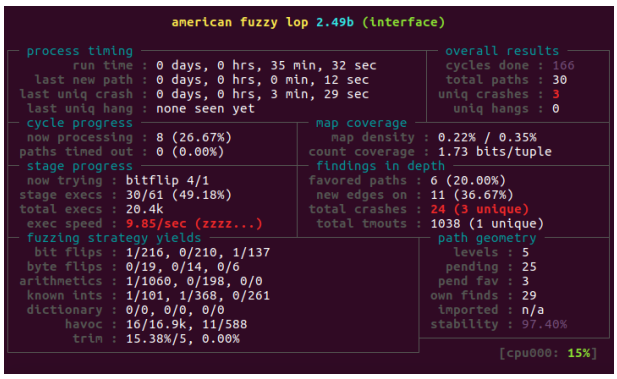

Figure 2: Snapshot of the AFL interface while fuzzing Apache Commons Imaging. The fuzzer has been running for 35 minutes, in which the target application was executed on 20400 different inputs. A total of 30 program behaviors have been explored, of which 3 result in a crash. While the paths leading to these 3 crashes are different, the cause is the same (`NegativeArraySizeException` in the `readBytes()` method in `org.apache.commons.imaging.common.BinaryFunctions`).

the library, a malicious user could cause a crash by sending a faulty JPEG[2].

A snapshot of the AFL interface after running Kelinci on Apache Commons Imaging for 35 minutes, presenting the results obtained so far, is shown in Fig. 2.

4 DISCUSSION

The design of the tool with a TCP connection and file IO adds overhead; furthermore running Java programs is much slower that running native programs. For example, on Apache Commons Imaging, Kelinci is approximately a factor 500 slower than AFL on C. Despite this, Kelinci is still useful in practice as evidenced by our findings which were obtained on a single machine. In addition, the autonomy of the tool—separating the fuzzer from the application being fuzzed—enables Kelinci to leverage a distributed infrastructure that would make it possible to perform scalable, parallel fuzzing; a powerful advantage of Kelinci. We intend to extend `interface.c` to connect to an array of servers, to enable this distributed approach.

Another limitation of Kelinci is that different runs are executed within the same Java Virtual Machine (JVM) in new threads. This assumes that the different runs are independent, i.e. they cannot influence each other. This is not true in general, as, for instance, different threads could access the same static locations. Even with only a single executor thread, static locations might not be properly reset. A more precise implementation would be to create a new JVM for each run, or to fork the entire JVM before starting the host program. Analogous to many design decisions in AFL, we take a *best effort* approach, where precision is traded against efficiency or, in this case, simplicity.

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

5 CONCLUSIONS

We have presented Kelinci, which, to the best of our knowledge, is the only currently viable option to apply AFL-style fuzzing to Java programs. It does not require modifications to the fuzzer, is highly parallelizable and has discovered a real bug in the popular image processing library, Apache Commons Imaging.

AFL has proved to be exceptionally successful at discovering security vulnerabilities. Compared to available tools for Java fuzzing, such as EvoSuite and Randoop, AFL operates at the system level while the other tools generate unit tests and is specifically targeted to security vulnerabilities. Kelinci opens up the possibility of testing Java based applications using this powerful technique.

Future Work. We intend to enable connecting the fuzzer side to an array of servers running Kelinci instrumented applications. This would enable running as many instances as desired in parallel in the cloud (the Java side can already accept requests from multiple fuzzer clients). It is expected that such parallelization creates a near-linear increase in performance.

We intend to use Kelinci as a platform for finding time/time vulnerabilities (notably Denial-of-Service). This research direction would comprise adding a *cost* dimension to AFL's logic for prioritizing inputs to fuzz, e.g., inputs that yield long computation times are more likely to be fuzzed.

Analogous to the work on Mayhem [2] and Driller [9], we also intend to combine Kelinci with symbolic execution, using Symbolic PathFinder [8].

REFERENCES

[1] ASM. 2017. http://asm.ow2.org/. (2017). Accessed August 11, 2017.
[2] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. 2012. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*. 380–394. https://doi.org/10.1109/SP.2012.31
[3] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 416–419.
[4] GCJ – GCC Wiki. 2017. https://gcc.gnu.org/wiki/GCJ. (2017). Accessed August 11, 2017.
[5] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages. https://doi.org/10.1145/2090147.2094081
[6] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279
[7] C. Pacheco, S.K. Lahiri, M.D. Ernst, and T. Ball. 2007. Feedback-directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 75–84. https://doi.org/10.1109/ICSE.2007.37
[8] Corina S. Pasareanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Autom. Softw. Eng.* 20 (2013), 391–425.
[9] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.
[10] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery*. Pearson Education.
[11] Michal Zalewski. 2017. American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl/. (2017). Accessed August 11, 2017.
[12] Michal Zalewski. 2017. Pulling JPEGs out of thin air. https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html. (2017). Accessed August 11, 2017.