JQF: Coverage-Guided Property-Based Testing in Java

Rohan Padhye
University of California, Berkeley
USA
rohanpadhye@cs.berkeley.edu

Caroline Lemieux
University of California, Berkeley
USA
clemieux@cs.berkeley.edu

Koushik Sen
University of California, Berkeley
USA
ksen@cs.berkeley.edu

ABSTRACT

We present JQF, a platform for performing coverage-guided fuzz testing in Java. JQF is designed both for *practitioners*, who wish to find bugs in Java programs, as well as for *researchers*, who wish to implement new fuzzing algorithms.

Practitioners write QuickCheck-style test methods that take inputs as formal parameters. JQF instruments the test program's bytecode and continuously executes tests using inputs that are generated in a coverage-guided fuzzing loop. JQF's input-generation mechanism is extensible. Researchers can implement custom fuzzing algorithms by extending JQF's `Guidance` interface. A `Guidance` instance responds to code coverage events generated during the execution of a test case, such as function calls and conditional jumps, and provides the next input. We describe several guidances that currently ship with JQF, such as: semantic fuzzing with Zest, binary fuzzing with AFL, and complexity fuzzing with PerfFuzz.

JQF is a mature tool that is open-source and publicly available. At the time of writing, JQF has been successful in discovering 42 previously unknown bugs in widely used open-source software such as OpenJDK, Apache Commons, and the Google Closure Compiler.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Coverage-guided fuzzing, property-based testing, QuickCheck

ACM Reference Format:
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 (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3293882.3339002

1 INTRODUCTION

Coverage-guided fuzzing (CGF) has recently become a very popular technique for automatic test-input generation. CGF tools like AFL [17] and libFuzzer [7] have discovered thousands of bugs and security vulnerabilities in programs that parse binary data, such as image decoders and media players. CGF works by first inserting lightweight instrumentation in a program under test for collecting code coverage. Then, the program is continuously executed with randomly generated inputs. If the program crashes, a bug is found. Instead of generating inputs from scratch, CGF evolves a set of saved inputs, starting with user-provided seed inputs. New inputs are generated via byte-level mutations such as bit flips and random insertion of interesting values. If a new input leads to an increase in code coverage, it is saved for subsequent mutation. The process repeats until a time budget expires.

```
1  @RunWith(JQF.class)
2  class TrieTest {
3    @Fuzz /* Arguments are generated randomly by JQF */
4    public void testMap2Trie(String key,
5                             Map<String,Integer> map){
6      assumeTrue(map.containsKey(key));
7      Trie trie = new PatriciaTrie(map); // Map2Trie
8      assertTrue(trie.containsKey(key));
9    }
10 }
```

Figure 1: A sample property test using JQF that checks the construction of a `Trie` data structure in Apache Commons from an input JDK `Map`. Fires an assertion violation (bug COLLECTIONS-714) when fuzzing with `ZestGuidance`.

The research community has produced several extensions to the basic CGF algorithm. Manès et al. [11] provide an extensive survey of various custom fuzzing algorithms found in the literature. For each of these new fuzzing algorithms, researchers produced standalone variants of AFL, libFuzzer, or other fuzzing tools.

In spite of all this research effort, most existing fuzzing tools target x86 binaries or C/C++ programs that expect inputs as binary or textual files. These tools are not suited for driving conventional software tests, where inputs can be arbitrary data structures. Property-based testing tools in the lineage of QuickCheck [3] allow randomized testing of such highly structured inputs, but don't support a feedback-directed fuzzing loop.

We present JQF, a platform for performing coverage-guided fuzzing of Java programs. The JQF platform meets the following design goals:

(1) JQF enables practitioners to use coverage-guided fuzzing for testing programs that require structured inputs, using the familiar style of property-based testing.
(2) JQF enables researchers to prototype new coverage-guided fuzzing algorithms to drive property-based tests.

JQF is publicly available at https://github.com/rohanpadhye/jqf as well as on Maven central (`edu.berkeley.cs.jqf:jqf-fuzz`).

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3339002

2 FUZZING WITH JQF

Practitioners can use JQF to automatically generate test inputs for parameterized test methods using coverage-guided fuzzing. Figure 1 shows an example of a JQF test driver written in Java, which aims to check a basic property of the class `PatriciaTrie` from Apache Commons Collections. A trie data structure can be constructed from a pre-existing mapping of strings in a JDK `Map` object.

The test method `testMap2Trie` checks the following property: Given an arbitrary string `key` and a JDK `map` whose keys are strings, if `key` exists within `map`, then a trie constructed from this map should also contain the same key. The `@Fuzz` annotation on the test method enables JQF to *automatically generate random instances* of `map` and `key` to verify this property. The JUnit `Assume` API allows the user to specify preconditions on the generated inputs (e.g. Line 6). Test generation can be launched via JQF's Apache Maven plugin[1]:

```
mvn jqf:fuzz -Dclass=TrieTest -Dmethod=testMap2Trie
```

By default, JQF uses the Zest algorithm (§4.2) to generate test inputs. Fuzzing continues either until it is explicitly stopped, until a user-specified timeout expires, or until a test failure is encountered.

For the test in Figure 1, the Zest fuzzing engine often finds a test failure in about 5 seconds, after executing about 5,000 test inputs (of which over 1,700 satisfy the precondition on Line 6). The failing test case leads to an assertion violation at Line 8 due to a very special corner case, which reveals a bug in Apache Commons Collections v4.3. If the input `map` contains two distinct keys that differ only in a trailing null character, say `"x"` and `"x\u0000"`, then the `trie` cannot distinguish between them and ends up storing only one of the two keys. If the input `key` is also `"x"`, then the bug is revealed[2].

JQF was specifically designed to enable practitioners to write test methods in the familiar style of property-based testing. JQF is built on top of `junit-quickcheck` [6], which itself is a Java port of the popular QuickCheck [3] tool. Thus, the test driver in Figure 1 can still be run with vanilla `junit-quickcheck`, which randomly generates test inputs without using code coverage feedback. However, random-from-scratch input generation is exceedingly unlikely to generate inputs fitting precise bug-revealing conditions, like those described above. Pure random generation does not find a failing test case for Figure 1 even after 30 minutes (over 7 million executions).

3 FUZZING FRAMEWORK

We next explain how JQF generates random inputs, such as `map` and `key` in Figure 1, using coverage-guided algorithms called *guidances*.

3.1 The `Guidance` Interface

Figure 2 shows the `Guidance` interface. Researchers can implement this interface to specify a coverage-guided fuzzing algorithm. `Guidance` instances are stateful objects whose methods are invoked by the JQF framework in a fuzzing loop (depicted in Figure 3).

The `Guidance` method `hasInput()` returns whether a new input is available; the return value `false` ends fuzzing. The `getInput()` method returns the next input generated by the `Guidance`, as an `InputStream`. This stream is used to generate structured inputs such as `Map` objects (see §3.2). The structured inputs, called `args` in Figure 3, are then used to execute the test method via JUnit (Line 7). Test execution generates `TraceEvents`, whose handling is described in §3.3. At the end of test execution, the `Guidance.handleResult()` method is invoked. The result can either be `SUCCESS`, `INVALID`, or `FAILURE`, depending on whether the test method returned normally (Line 8), due to a violation of `assume` (Line 10), or due to an exception/assertion violation (Line 12), respectively. The `Guidance` instance updates its internal state based on the handling of code coverage events and the test result. The internal state is then used to generate new inputs in subsequent iterations of the fuzzing loop.

```
1  public interface Guidance {
2      boolean hasInput();
3      InputStream getInput();
4      void handleResult(Result result, Throwable error);
5      Consumer<TraceEvent> generateCallBack(Thread thread);
6  }
```

Figure 2: The `Guidance` interface provided by JQF.

```
1  TestMethod test   = ...; // @Fuzz test driver
2  Guidance guidance = ...; // Fuzzing algorithm
3  while (guidance.hasInput()) {
4      // Generate args for test method
5      Object[] args = JQF.gen(test, guidance.getInput());
6      try {
7          JUnit.run(test, args); // fires TraceEvent(s)
8          guidance.handleResult(SUCCESS, null);
9      } catch (AssumptionViolatedException e) {
10         guidance.handleResult(INVALID, e);
11     } catch (Throwable t) {
12         guidance.handleResult(FAILURE, e);
13     }
14 }
```

Figure 3: Pseudo-code of the fuzzing loop.

3.2 Parametric Generators

The arguments to a test method—such as `map` and `key` in Figure 1—are generated using the same mechanisms as supported by `junit-quickcheck`. In general, inputs of type `T` are generated by a backing `Generator<T>`, which provides a method to *randomly sample a new instance of* `T`. `junit-quickcheck` can either (1) implicitly pick a suitable generator from a library that it provides, (2) be directed to synthesize such a generator automatically, e.g. using the constructors or public fields of class `T`, or (3) be provided with a hand-written `Generator<T>`.

In all cases, the generator uses a `SourceOfRandomness` object, which provides an API for making non-deterministic decisions such as: choosing from a list of alternatives (e.g. whether to instantiate a `TreeMap` or `HashMap` for `map` in Figure 1), picking random sizes (e.g. how many entries to insert in `map`), or populating primitives (e.g. what keys and values to insert in `map`). In `junit-quickcheck`, the default `SourceOfRandomness` is backed a pseudo-random stream of bytes. JQF overrides this source to use the stream returned by `Guidance.getInput()` instead (ref. Line 5 in Figure 3), thereby making the generators deterministically dependent on the guidance.

[1] Non-Maven users can launch JQF programatically or via command-line scripts.
[2] We stumbled upon this bug while writing an example for this paper.

3.3 Code Coverage Events

When coverage-guided fuzzing is launched (e.g. via `mvn jqf:fuzz`), the test program's classes are instrumented on-the-fly using the ASM bytecode manipulation library [13]. The instrumentation adds logic to generate `TraceEvents` during test execution. For example, a `BranchEvent` is generated when a test program executes a conditional branch, a `CallEvent` accompanies a method invocation, and an `AllocEvent` signals the creation of a new object or array on the heap. These event objects contain information about their source program locations as well other event-specific data. When a trace event `e` is generated in thread `t`, JQF invokes the function `handle_t(e)`, where `handle_t` is the callback returned by `Guidance.generateCallBack(t)`. The guidance must choose how to update its internal state based on this coverage information, which will presumably be used to generate subsequent inputs.

4 GUIDANCES

JQF currently ships with the following `Guidance` implementations.

4.1 No Guidance

The most trivial guidance, called `NoGuidance`, returns an infinite stream of random values every time `getInput()` is called. This guidance completely ignores code coverage events. This guidance is almost equivalent to using vanilla `junit-quickcheck`.

4.2 Zest Guidance

JQF's default guidance implements the Zest algorithm [15], which is specifically designed for coverage-guided property testing. The `ZestGuidance` returns dynamically sized *parameter sequences* via the `getInput()` method, which are generated randomly for the first iteration of the fuzzing loop. Zest maintains a set of saved parameter sequences. The `ZestGuidance` generates new inputs by randomly mutating previously saved parameter sequences. Byte-level mutations on these parameter sequences correspond to structural mutations in the generated test inputs. For example, a random mutation in the parameter sequence for `map` in Figure 1 may lead to the corresponding `Generator<Map>` to produce the next map with an additional entry. Dynamic sizing allows the parameter sequences to be lazily extended (if the `Generator` needs to make more choices than expected) or to be efficiently truncated (if the `Generator` makes fewer choices). Further, Zest separately tracks code coverage achieved by *all* test executions and code coverage by *valid* test executions (i.e., those whose result is `SUCCESS`). If a mutated parameter sequence leads to new code coverage overall, or if it leads to a valid test that covers code which has not been covered by any previous *valid* test, then the sequence is saved for subsequent mutation. Zest has been used to find complex semantic bugs, such as issues within compiler optimizations[3].

4.3 AFL Guidance

JQF supports input generation using the popular AFL [17] tool, unmodified. This is possible because AFL, which is designed to fuzz C/C++ programs and x86 binaries, communicates with instrumented test programs via inter-process messages and a code coverage map in shared memory. The `AFLGuidance` in JQF implements this communication protocol via a proxy program. The proxy mocks an AFL-instrumented test target that reads input from a specific file. `AFLGuidance.getInput()` simply returns the contents of this file, which is continuously updated by AFL. During test execution, `AFLGuidance` collects code coverage information by handling `TraveEvents`. When `AFLGuidance.handleResult()` is invoked, the coverage information is written to AFL's shared memory region via the proxy. Calls to `AFLGuidance.hasInput()` block until AFL is ready with the next input.

AFL's mutation strategy uses various heuristics that are applicable to programs that parse fixed-size binary files (e.g. media players). Further, AFL does not explicitly distinguish between `INVALID` and `FAILURE` results. Due to these reasons, JQF's `AFLGuidance` is most effective when used with test methods that take only one argument of type `InputStream` (since `Generator<InputStream>` returns the guidance-generated input stream as-is), and that do not use any `assume` statements. For example, `AFLGuidance` has been used to fuzz OpenJDK's ImageIO library that reads PNG and JPEG files[4], as well as Apache PDFBox's processing of PDF documents[5].

4.4 PerfFuzz Guidance

PerfFuzz [10] is a technique for automatically generating test inputs that maximize performance counters, such as loop execution counts. PerfFuzz's goal is to automatically discover hot spots and performance bottlenecks. PerfFuzz is a fork of AFL that extends its code coverage map with performance feedback in the form of $\langle k, v \rangle$ pairs where v is a value to be maximized for every key k. PerfFuzz saves a mutated input either if it leads to new code coverage, or if it maximizes the value of v for some key k.

JQF's `PerfFuzzGuidance` is a sub-class of `AFLGuidance` which overrides `handleResult()` to communicate this additional performance map via the proxy program. `PerfFuzzGuidance` can be configured either to find hot spots (where keys are branch locations and values are execution counts for the corresponding branch) or to find memory consumption issues (where keys are allocation sites and values are number of bytes allocated at the corresponding site). For example, we used `PerfFuzzGuidance` to find an algorithmic complexity bug in the Google Closure Compiler, where reporting a specific case of syntax error in a JavaScript program can take time that is exponential in the size of the input program[6]. With the memory allocation feedback, we found an issue in OpenJDK's handling of PNG images that specify very large dimensions[7].

4.5 Repro Guidance

Finally, the `ReproGuidance` is a trivial guidance whose `getInput()` method returns the contents of a given file on disk, and then ends the loop. This guidance enables debugging of saved test failures.

5 EVALUATION AND IMPACT

Table 1 summarizes the impact that JQF has had in discovering previously unknown bugs in widely used Java software. These

[3] https://github.com/google/closure-compiler/issues/2842

[4] https://bugs.openjdk.java.net/browse/JDK-8191073
[5] https://issues.apache.org/jira/browse/PDFBOX-4333
[6] https://github.com/google/closure-compiler/issues/3173
[7] https://bugs.openjdk.java.net/browse/JDK-8190332

Table 1: Number of new bugs discovered using JQF.

Project	Bugs Found	Bugs Fixed
OpenJDK - ImageIO	9	9
OpenJDK - DateTime	2	1
Apache Commons - Lang	1	1
Apache Commons - Compress	2	2
Apache Commons - Collections	1	0
Apache Maven	3	3
Apache Ant	1	1
Apache BCEL	8	0
Apache PDFBox	4	4
Apache Tika	2	2
Google Closure Compiler	4	1
Mozilla Rhino	5	0
Total	42	24

bugs were found over the course of various experiments performed throughout 2017–2019. The Zest paper [15] describes a systematic study involving five of the projects from this table and three different guidances. The study showed that `NoGuidance` is not very reliable, that `AFLGuidance` is effective in finding bugs in syntax parsers, and that `ZestGuidance` excels at finding semantic bugs. This study resulted in the discovery of 20 of the bugs in Table 1.

Of the total 42 bugs found using JQF, 11 semantic bugs were found with `ZestGuidance`, 29 syntax parsing bugs were found with `AFLGuidance`, and 2 bugs were found with `PerfFuzzGuidance`. 24 of the 42 reported bugs have been fixed at the time of writing, while the rest await patches.

Notably, 7 of the 42 bugs (including 4 security vulnerabilities with assigned CVEs) were discovered by two independent practitioners who are not affiliated with the authors of this paper. We were made aware of JQF's success via social media [1] and blog posts [12]. All 7 of these bugs have been fixed. We are encouraged by these findings, and believe that they provide evidence to support JQF's usefulness to the software testing community at large.

6 RELATED WORK

JQF is one of few tools that enable coverage-guided fuzzing of Java programs. Kelinci [9] is a wrapper around `afl-fuzz` that targets Java programs. Unlike JQF's extensible guidance mechanism, Kelinci's instrumentation directly updates AFL-specific coverage feedback; therefore, it cannot easily be adapted to work with tools like PerfFuzz. Kelinci also expects a test driver with a `main` method that reads inputs as files, in contrast to JQF's property-based testing approach. Thus, Kelinci does not support structured input fuzzing algorithms such as Zest. Further comparison between Kelinci and JQF can be found in a blog post by an independent security company [12]. Barro [8] has implemented another wrapper around AFL for fuzzing Java `main` programs, similar to Kelinci. This tool actually borrows its dynamic instrumentation logic from JQF itself.

There exist several other tools for generating tests for Java programs. Randoop [14] generates method call sequences randomly, but does not use code coverage feedback. Evosuite [4] evolves a test suite using genetic algorithms. Java PathFinder [16] enables symbolic execution of Java programs. Korat [2] systematically explores the input space of Java predicates. UDITA [5] performs bounded exploration of test-input generators. Our tool, JQF, differs mainly in that supports a different mechanism for test-input generation: that of coverage-guided fuzzing.

ACKNOWLEDGMENTS

We would like to thank Tobias Ospelt of modzero AG for reporting various issues with alpha versions of JQF. We appreciate the feedback of Tim Allison, VP of Apache Tika, on early bugs found by JQF. We are also grateful to Paul Holser for developing the very extensible `junit-quickcheck` library, using which JQF is built. This research is supported in part by gifts from Samsung, Facebook, and Futurewei, and by NSF grants CCF-1409872 and CNS-1817122.

REFERENCES

[1] Tim Allison. 2018. #threeCheersForFuzzing. https://twitter.com/_tallison/status/1050455776848949249. Accessed April 17, 2019.
[2] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 123–133. https://doi.org/10.1145/566172.566191
[3] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*.
[4] 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 (ESEC/FSE '11)*.
[5] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. 225–234. https://doi.org/10.1145/1806799.1806835
[6] Paul Holser. 2014. junit-quickcheck: Property-based testing, JUnit-style. https://pholser.github.io/junit-quickcheck. Accessed January 11, 2019.
[7] LLVM Compiler Infrastructure. 2016. libFuzzer. https://llvm.org/docs/LibFuzzer.html. Accessed April 17, 2019.
[8] Jussi Judin. 2018. Binary rewriting approach [...] to fuzz Java applications with afl-fuzz. https://github.com/Barro/java-afl. Accessed April 17, 2019.
[9] 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*. ACM, 2511–2513.
[10] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874
[11] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2018. Fuzzing: Art, Science, and Engineering. *CoRR* abs/1812.00140 (2018). arXiv:1812.00140 http://arxiv.org/abs/1812.00140
[12] Tobias Ospelt. 2018. AFL-based Java fuzzers and the Java Security Manager. https://www.modzero.ch/modlog/archives/2018/09/20/java_bugs_with_and_without_fuzzing/index.html. Accessed April 17, 2019.
[13] OW2 Consortium. 2018. ObjectWeb ASM. https://asm.ow2.io. Accessed August 21, 2018.
[14] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA '07)*.
[15] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. https://doi.org/10.1145/3293882.3330576
[16] Willem Visser, Corina S PÇÕsÇÕreanu, and Sarfraz Khurshid. 2004. Test input generation with Java PathFinder. In *ACM SIGSOFT Software Engineering Notes*, Vol. 29. ACM, 97–107.
[17] Michał Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl. Accessed January 11, 2019.