Fuzz Testing for Apache Spark XML Document Parser Using JQF + Zest

Jordan Sam and Benson Wan

I. INTRODUCTION

When it comes to software, testing is critical to saving time and money. We test software to find bugs, reduce risk, and improve quality. By providing quality software and finding bugs early, we can strengthen consumer satisfaction and make everyone's lives easier. Although it is expensive and time-consuming to initially test, not testing is even more expensive [1]. Today, a technique called coverage-guided fuzz testing has been increasing in popularity for generating test-inputs automatically. Coverage-guided fuzzing instruments a program to trace the code coverage that generated inputs reach. We have seed inputs that are provided by the user and we generate new inputs through byte level mutations. A fuzzing engine then makes decisions on what inputs to mutate and feed the randomly generated inputs back into the program. These inputs can lead to a bug if the program crashes or it can increase coverage. If one of these two things stated above occurs, we save the input and mutate it again to repeat the process and continue our testing. The platform we are using to conduct coverage-guided fuzz testing in our project is JQF and Zest. Our software under test is Apache Commons and specifically the Apache Maven pom.xml file parser. The inputs that the program takes in are generated within the xml and commons folder that contain the input generators themselves. If you are interested in recreating the Apache Maven bug you can review the code at https://github.com/Jsam88/CS182-Project.

II. TECHNICAL APPROACH

In JQF we label each location of java instructions at the back code level and instrument software bytecode. JQF is represented as a list of trace events and its complexity is reduced by utilizing a coarse-grained system. An example of a trace event includes when a program under test executes a conditional branch, the BranchEvent is generated. Another example is the AllocEvent which is generated when a new object is created. It is important to further note that many programs under test usually consist of inputs that are both syntactically and semantically valid. The syntax parsing allows programs to run properly whereas the semantic analysis covers the logic of the input.

A. Test Generation

In our software testing tool, we utilize a structured input generator to cover the semantics of our program through junit-quickcheck and the zest algorithm. Junit-quickcheck is the library for property based tests and for syntactically valid inputs. Using the testWithGenerator() function, a random XML document can be generated. From there, the function testWithInputStream can be used to generate the root element of the randomly generated XML document. Using the testProgram method, the random XML document will be serialized and will invoke the ModelReaderTest method to parse the XML document into its appropriate internal model. If the ModelReaderTest method is successful, one of the core functionalities of that program will be tested and the testWithGenerator method will be fulfilled. However, if there is a syntactic or semantic error when the ModelReaderTest function is run, the program fails. Essentially, the program works by creating a random generation, checking the validity of that generation, then running the necessary functions with that random generation.

B. Coverage Guidance

The zest algorithm is utilized for code coverage and input validity feedback to help quickcheck. This basically guides quickcheck to reach semantics for the program under test. Zest is able to incorporate the power that coverage-guided fuzzing brings into generator-based testing through acquiring a deterministic parametric generator that is equivalent to what was once a randomized-input generator before the transformation. Zest then proceeds to search through the parameter space through a process called feedback-directed parameter search. This process of searching is able to work in tandem with the coverage guided fuzzing algorithm through making note of the code coverage that valid inputs were able to accomplish. Ultimately, deeper paths of code are able to be reached through the use of this technique which enables for a more fruitful semantic analysis stage

```
@RunWith(JQF.class)
public class ModelReaderTest {

    2 usages  Jsam88
    @Fuzz
    public void testWithInputStream(InputStream in) {
        ModelReader reader = new DefaultModelReader();
        try {
            Model model = reader.read(in, null);
            Assert.assertNotNull(model);
        } catch (IOException e) {
            Assume.assumeNoException(e);
        }
    }

    1 usage  Jsam88
    @Fuzz
    public void testWithGenerator(@From(XmlDocumentGenerator.class)
                                  @Size(min = 0, max = 10)
                                  @Dictionary("maven-model.dict") Document dom) {
        testWithInputStream(XMLDocumentUtils.documentToInputStream(dom));
    }

    Jsam88
    @Fuzz
    public void debugWithGenerator(@From(XmlDocumentGenerator.class)
                                   @Size(min = 0, max = 10)
                                   @Dictionary("maven-model.dict") Document dom) {
        System.out.println(XMLDocumentUtils.documentToString(dom));
        testWithGenerator(dom);
    }
```

Figure 1: A sample property test using JQF that checks the construction of a Model structure in Apache Commons from an Apache Spark XML Document input

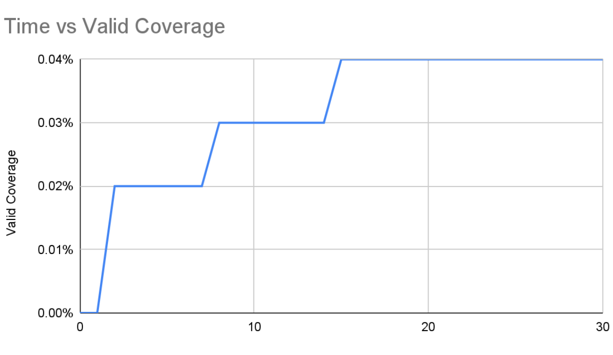

Figure 2: Time vs valid coverage graph. After 15 seconds, we stop furthering our coverage of our program

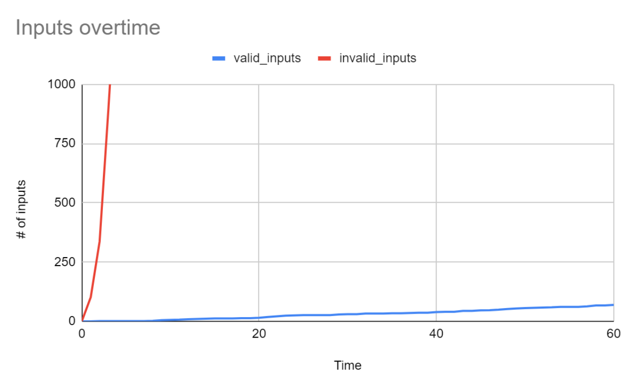

Figure 2: Time vs inputs graph. We see the # of invalid inputs are exponentially generated faster than the valid inputs.

III. EVALUATION

Our evaluation seeks to answer the following questions:
RQ1 How much improvement is whitebox testing compared to blackbox testing (blind fuzzing)?

RQ2 Are there any differences between running this program on a different operating system?

A. Analysis

Our results were acquired from a wide range of tests which were run through a multitude of functions such as testWithString, testWithInputStream, testWithGenerator, and testWithGenerator blind. The branches covered for each test result were always 28-29 branches with the exception of one being around 31 branches and two ranging from 0-2 branches covered. Along with the branches that were reached, almost every test showed results of having the total coverage and valid coverage be equivalent with one another. The average number of executions across our results came out to be 28,687,318 across the board of ten different tests. All of the raw data of our results can be found here at https://github.com/Jsam88/CS182-Project/tree/main/model-read-test/target/fuzz-results/examples.ModelReaderTest.

B. Behavior (blackbox vs whitebox)

While testing our software, we decided to utilize both blackbox testing and whitebox testing. We are able to see that with blackbox testing we have a higher execution rate. This is because the inputs are more random and not guided by the previous inputs at all. This ultimately leads to us having a lot less valid inputs compared to whitebox testing but we end up having the same total/valid coverage.

C. Differences between Operating Systems

We tested our program in 2 different operating systems with them being MacOS and Windows. Starting with MacOS, we ran into multiple issues. Although installing JQF and getting other toy programs running and fuzzing, we ran into errors with our software under test (Apache Maven) and it can be seen in figure 2.


```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  5.254 s
[INFO] Finished at: 2022-11-21T14:37:45-08:00
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal edu.berkeley.cs.jqf:jqf-maven-plugin:1.8:fuzz (default-cli) on project jqf-zest-example: Internal error: Too many trials without coverage; likely all assumption violations -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoExecutionException
```

Figure 2 Error on MacOS (Too many trials without coverage)

Our first attempt to solve this problem was to remove the try and catch block in one of the JQF libraries. However we discovered shortly that the file is only read only which makes it uneditable. After trial and error we were able to work around this issue by downloading ubuntu for windows and running the program through there. Although both our MacOS and windows environments had the same java and maven version installed, running ModelReadTest was successful only on windows with ubuntu.

D. Optimization

Although we ran JQF and fuzzed the program for over 70 hours collectively, we were not able to recreate the apache maven bug. The limitations of our tests were not having enough time to generate more inputs. With more time, we may have been able to recreate the apache maven bug. However, this is a real world problem that we are trying to solve. We have a finite amount of time and money that can be spent on programming and testing software quality. We can cut some of the costs by improving the generated tests. This is done by manually creating inputs that cover the map of the program better. Since our failures may be more subtle, we can create generic checkers to catch more errors [5]. This involves memory leaks, information leaks, runtime errors, and more. One of the issues that Rohan Padhye ran into was the length of the XML file being too big to create inputs for mavens parsing logic. Similarly, we can add more logic and attempt to reach the main program and have both syntactically and semantically meaningful inputs.

IV. COLLABORATION

Jordan:

- Created presentation
- Collected data/results
- Created visuals to draw conclusions from
- Wrote introduction, technical approach and analyzed results.
- Compared MacOs vs Windows error
- Worked on analyzing blackbox vs whitebox
- Compared operating systems.

Benson:

- Created presentation
- Analyzed data/results for visualization purposes
- Analyzed zest and coverage guided fuzzing algorithm
- Compared operating systems
- Wrote introduction, technical approach, and evaluation

REFERENCES

[1] Qian Zhang, Lecture 2, "Testing Overview"
[2] R. Padhye, C. Lemieux, and K. Sen, "Jqf: Coverage-guided property based testing in java," in Proceedings of the 28th ACM SIGSOFT Interna tional Symposium on Software Testing and Analysis, ISSTA 2019, (New York, NY, USA), p. 398–401, Association for Computing Machinery, 2019.
[4] https://github.com/rohanpadhye/jqf/wiki/Fuzzing-with-Zest, 2021.
[5] https://www.fuzzingbook.org/html/Fuzzer.html. 2022.