jFuzz: A Concolic Whitebox Fuzzer for Java

Karthick Jayaraman
Syracuse University
kjayaram@syr.edu

David Harvison, Vijay Ganesh, Adam Kieżun*
MIT
{dharv720, vganesh, akiezun}@mit.edu

Abstract

We present jFuzz, a automatic testing tool for Java programs. jFuzz is a concolic whitebox fuzzer, built on the NASA Java PathFinder, an explicit-state Java model checker, and a framework for developing reliability and analysis tools for Java. Starting from a seed input, jFuzz automatically and systematically generates inputs that exercise new program paths. jFuzz uses a combination of concrete and symbolic execution, and constraint solving.

Time spent on solving constraints can be significant. We implemented several well-known optimizations and name-independent caching, which aggressively normalizes the constraints to reduce the number of calls to the constraint solver. We present preliminary results due to the optimizations, and demonstrate the effectiveness of jFuzz in creating good test inputs. The source code of jFuzz is available as part of the NASA Java PathFinder.

jFuzz is intended to be a research testbed for investigating new testing and analysis techniques based on concrete and symbolic execution. The source code of jFuzz is available as part of the NASA Java PathFinder.

1 Introduction

We present jFuzz, a concolic whitebox fuzzer for Java built on top of the NASA Java PathFinder (JPF) [4]. jFuzz takes a Java program and a set of inputs for that program. For each input, jFuzz creates new inputs that are modified (or *fuzzed*) versions of the input and exercise new control paths in the program.

jFuzz (similarly to other concolic whitebox fuzz testing tools [7, 11]) executes the program both concretely and symbolically [7, 9, 14]. jFuzz converts the symbolic execution into a logical formula called a *path constraint*. jFuzz systematically negates every conditional along the execution path, conjoins the conditional with the corresponding path constraint, and queries a constraint solver. The solution, if one exists, is in terms of values for parts of the input. jFuzz uses the solution to fuzz (modify) these parts to obtain a new input. The appropriately fuzzed inputs can thus explore previously unexamined branches along the execution path. Thus, jFuzz can systematically explore every control-flow path.

The time spent in constraint solving can be significant [11], because (i) constraints may be hard to solve, or (ii) the solver may be repeatedly solving a large number of very similar problems. We implemented well-known optimizations such as constraint caching, constraint independence and subsumption [9, 11, 14] that seek to simplify the interaction of the testing tool with the solver. In addition, we also implemented name-independent caching, a new optimization that aggressively normalizes path constraints generated during concolic execution. This technique detects equivalence between two constraints modulo variable renaming. Thus, jFuzz caches solutions to already-solved constraints, and whenever jFuzz detects an equivalence between as yet unsolved constraint and an already-solved constraint, the cached solution is denormalized and reused, thus reducing the number of calls to the solver.

Contributions

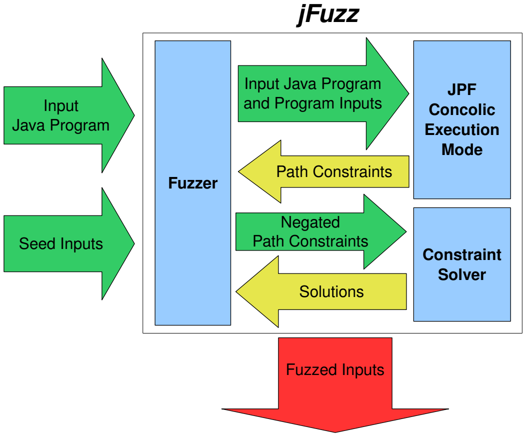
- **Concolic execution mode in JPF:** Concolic execution combines concrete and symbolic execution. Having this mode implemented in a reliable open-source framework such as JPF will facilitate further research in systematic software testing. The source code of the concolic execution mode is available as part of JPF.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 121-125

*To whom correspondence should be addressed.

```
public static int modtenadd(int x, Int y)       {

 int z;

 if(x >= 10 || y >=10)
 {
        throw new InvalidArgumentException();
 }

 z = x + y;

 if(z >= 10)
 {
        z = z - 10;
}
return z;
}
```

(a) A Java function that performs modulo 10 addition



(b) jFuzz architecture. Given the program under test and seed inputs, jFuzz generates new inputs by modifying (fuzzing) the seed inputs so that each new input executes a unique control-flow path.

- **jFuzz:** jFuzz is a concolic whitebox fuzzer for Java, built on top of the JPF's concolic execution mode (which can be used independently of jFuzz). jFuzz is intended as a research vehicle for development of smart fuzzing techniques. The source code of the concolic execution mode is available as part of JPF.

- **Experimental Evaluation:** We present preliminary experimental results. We evaluated the efficiency of the concolic execution mode, effectiveness jFuzz's, and the performance of name-independent caching. In our experiments, the concolic mode added only 15% overhead above normal JPF execution. Tests created by jFuzz achieved slightly higher coverage than random fuzzing (18% vs. 15% line coverage). The constraint optimizations that we added reduced the time spent in constraint solving that ranged between 25%-30% to 1%.

2 Example

We illustrate whitebox fuzzing on an example (Figure 1(a)) Java function that performs modulo-10 addition on two integers. To start whitebox fuzzing, we provide the program and an initial concrete input $x = 3, y = 4$ to the fuzzer. Concolic execution produces a result of $z = 7$ and symbolic constraints $(x < 10) \bigwedge (y < 10) \bigwedge (x + y < 10)$.

Now, the whitebox fuzzer sequentially inverts each constraint and produces three sets of constraints.

1. $(x < 10) \bigwedge (y < 10) \bigwedge (x + y >= 10)$
2. $(x < 10) \bigwedge (y >= 10)$
3. $(x >= 10)$

The whitebox fuzzer solves these constraints using a constraint solver and obtains three new inputs $(6,6), (3,11), (11,3)$. Each of the three generated inputs exercises a distinct execution path in the program. The whitebox fuzzer repeats the process with the new inputs for either a pre-specified number of executions or time duration.

3 jFuzz Overview

jFuzz is built on top of the NASA Java PathFinder framework [4]. The Java PathFinder is an explicit state software model checker for Java bytecode, that also provides hooks for a variety of analysis techniques. Figure 1(b) illustrates the architecture of jFuzz. jFuzz works in three steps:

1. Concolic Execution: jFuzz executes the subject program in the concolic-execution mode on the seed input, and collects the path constraint. Each byte in the seed inputs is marked symbolic. The path constraint is a logical formula that describes the set of concrete inputs that would execute the same control-flow path as the seed input.

2. Constraint Solving: Once the concolic execution has completed, jFuzz systematically negates the conditionals encountered on the executed path. jFuzz conjoins the corresponding path constraint with the negated conditional, to obtain a new constraint query for the solver. The solution is in terms of input bytes, i.e., describes the values of the input bytes.

3. Fuzzing: For each solution, jFuzz changes the corresponding input bytes of the initial seed input to obtain a new fuzzed input for the program under test.

3.1 Concolic Execution Mode in Java PathFinder

One of the contributions of this paper is the concolic execution mode in JPF. This mode can be used independently of jFuzz, to construct new research tools that employ concolic techniques. The concolic mode is inspired by the symbolic execution mode already available in Java PathFinder [12]. JPF provides the facility to associate *attributes* with runtime values. Concolic (and symbolic) execution mode uses attributes to associate symbolic constraints with runtime values, and extends the Java bytecode instructions to update the symbolic constraints during concolic execution. The differences between concolic and symbolic mode are:

- Concolic mode preserves concrete values for runtime values, while symbolic mode loses them.
- Concolic mode does not fork and backtrack the execution. This improves performance because it does not require state matching. Debugging concolic execution is also much simpler.

3.2 Name-Independent Caching

Path Condition 1:	Path Condition 2:
[1] a + b < 10	[1] x + y < 10
[2] b > 6	[2] y > 6
[3] a < 3	[3] x < 3
[4] a != 2	[4] x != 2

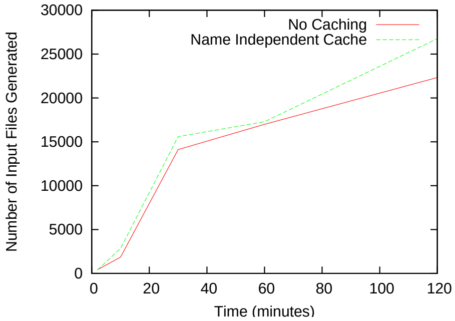
Figure 1: Two path conditions that are equivalent under name-independent caching.

A key issue with whitebox fuzzing is that the cumulative time spent in constraint solving can be a signification percentage of the time taken for producing new fuzzed inputs [11].

In jFuzz, we implemented several well-known optimizations such as constraint caching, constraint independence and subsumption [9, 11, 14]. Additionally, we implemented name-independent caching, an optimization that normalizes path constraints generated during concolic execution. This technique detects equivalence between two constraints modulo variable renaming. Thus, solutions to already-solved constraints are cached, and whenever equivalences between as yet unsolved constraints and already-solved constraints are detected, the cached solutions are denormalized and reused, resulting in reduced number of calls to the solver. For example, consider the two path constraints in Figure 1. The name-independent cache detects that the two path constraints are structurally equivalence

		coverage	
mode	inputs	line	block
Seed inputs	15	13%	12%
Random fuzzing	300	14%	13%
jFuzz	264	18%	17%

(a) Coverage results for 1-hour test-runs.



(b) Number of input files generated with and without name-independent caching in a given amount of time.

Figure 2: Experimental Results

and passes only one to the constraint solver. Without this optimization, a fuzzing tool redundantly calls the constraint solver for both constraints.

4 Experimental Evaluation

We evaluated the efficiency of the concolic execution mode, jFuzz's effectiveness, and the performance of name-independent caching.

Experimental Setup. As a subject program, we used SAT4J [5], a Boolean SAT solver (19419 lines of Java code). We obtained 15 seed inputs from the SAT competition website [3] All experiments were performed on an Intel Centrino Duo 1.4 GHz processor with 2GB RAM running the GNU/Linux operating system (Ubuntu version 8.04). Each testing technique (or testing mode) was given 1 hour testing time. We measured line and block coverage using the Emma tool [2]. The time required by the random fuzzer and jFuzz to construct their test suites is included in the testing time. Consequently, both the random fuzzer and jFuzz had lower timeout per test.

Results. The concolic mode adds only modest overhead to JPF. We compared the execution times of 18 test programs when executed using the JPF concolic execution mode, Java PathFinder, and the Java virtual machine. Compared to the Java virtual machine, Java PathFinder (in simulation mode) has an average slow down of 12×, while the concolic execution mode has an average slow down of 14×.

jFuzz creates effective tests. Inputs generated by jFuzz achieve better coverage than randomly generated inputs and markedly increase coverage from seed inputs (Table 2(a)).

The optimizations are effective. The proportion of time spent in constraint solving ranged between 25% to 30% without optimizations, compared to 1% with the optimizations. Also, the caching increased the number of generated files, per unit of time, by 16% (Figure 2(b)). The optimizations reduced the number of redundant calls to the constraint solver and enables jFuzz to spend more time on generating new input files. The cache hit-rate, depending on the example, ranged between 30% and 50%.

5 Related Work

A number of testing tool are based on combined concrete and symbolic execution [1, 6, 7, 8, 9, 11, 13, 14]. However, as far as we know, only CREST [1] (a tool for testing C programs) is publicly available. Furthermore, most of these tools are *end-user* tools, and thus not necessarily extensible by other researchers.

jFuzz builds on top of extensible and mature technology, Java PathFinder explicit-state software model checker and dynamic-analysis framework. We believe that jFuzz will provide an extensible platform for researchers to try new concolic-based reliability techniques.

jFuzz's name-independent caching is a simple yet effective technique to reduce the cumulative time spent in constraint solver. Testing tools use constraint caching [7, 8, 10, 11], and other optimizations such as syntactic subsumption [11], and unrelated constraint elimination [9, 14]. However, as far as we know, the name-independent caching scheme that we implemented in jFuzz has not been used before in systematic testing.

jFuzz represents work in progress, and our results are preliminary. We plan to use jFuzz to test many larger Java programs. We believe that other researchers will find jFuzz useful as a testbed to try new concolic-based techniques.

Acknowledgments

We thank Corina Păsăreanu and Peter Mehlitz from the JPF team for their assistance with the development of the concolic mode and in getting jFuzz included in Java Path Finder.

References

[1] CREST: automatic test generation tool for c. `http://code.google.com/p/crest`.

[2] EMMA: A Java code coverage tool. `http://emma.sourceforge.net/`.

[3] The international SAT competitions web page. `http://www.satcompetition.org/`.

[4] NASA Java PathFinder. `http://javapathfinder.sourceforge.net`.

[5] SAT4J: A Java sat solver. `http://www.sat4j.org/`.

[6] Shay Artzi, Adam Kieżun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael Ernst. Finding bugs in dynamic Web applications. In *ISSTA*, 2008.

[7] Christian Cadar, Vijay Ganesh, Peter M. Pawlowski, David Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *CCS*, 2006.

[8] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, 2005.

[10] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Active property checking. In *EMSOFT*, 2008.

[11] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[12] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, 2008.

[13] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. *LECTURE NOTES IN COMPUTER SCIENCE*, 4144:419, 2006.

[14] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *FSE*, 2005.