Javascript Fuzzing

Prabhat Gehlot, Sarkis Mikaelian, Thomas Perez
https://github.com/citationdude/410project
California State University Northridge
Kyle Dewey Ph.D.

Abstract

Fuzzing is a popular, and performant automated software testing technique to test, and assess the quality of complex software[3]. It involves feeding semi-random programs into a software system and recording the system's behaviour. Our project aims to utilize the inherent non-determinism, and constraint bounds of Constraint Logic Programming to generate programs for fuzz testing. We use *prolog* to implement our approach, and target *Javascript* engine as our software system to illustrate this paradigm of test-case generation.

Introduction

Traditionally, Fuzz testing has been implemented using dynamic, and object oriented programming languages: JSFuzz, a coverage-guided fuzzer for javascript and Node.js packages was written in Javascript; JSFunFuzz, another fuzzer for testing the javascript engine was implemented in python[4].

CLP's inherent non-determinism, and its efficient design of the backtracking procedure makes it a fruitful ground for test case generation. In our context, we define the test case in BNF grammar -- valid javascript program definitions that the engine should accept -- and use prolog's internal backtracking implementation to generate valid test cases. As an extension, we also explore parameters, implemented in prolog, that can offer greater control over the cases generated. In particular, we implement bound control which limits the depth of each abstract syntax tree, and randomness in the program generated.

Related Work

In the economics of software testing, fuzz testing has proven to be an efficient and reliable way to test complex software, while building confidence in the underlying logic of the system.

JSFunFuzz, implemented by independent researchers, was able to find 280 bug in the spidermonkey engine at the time of its release; similar fuzzers have shown promising results in testing.

Dewey et. al. were able to successfully implement test case generation using CLP. In part, our paper tries to replicate a subset of the test features tested by Dewey et. al. for the Javascript engine.

Application

We utilize the BNF definition of Javascript syntax to define valid programs in our prolog logic base. An example of a valid expression is mentioned in figure 1.

[Figure 1: An expression]

Using the definition listed above, we translate it into prolog as described in figure 2.

[Figure 2: Expression from figure 1 in prolog]

It is not difficult to see how a valid BNF grammar can be translated into prolog. With the BNF expression from figure one defined, if we query our prolog logic base with, *exp(X).* It will respond with all valid expressions as defined by the grammar in figure 1. With this simple construction we define a variety of Javascript syntax including but not limited to switch statements, for loops, while loops and many others.

Backtracking in the prolog interpreter is performed sequentially. And with nested self-referencing expressions, the sequential interpretation results in a boundless expansion of the search space in one direction only. To overcome this limitation, we employ a bound which limits the expansion in any branch of the search tree by a bound. Additionally, to generate interesting programs, we traverse the search space randomly: meaning that, we explore the branches in a random order instead of a well-defined sequence.

Evaluation

We implement some basic *javascript* syntax in prolog using the methodology described above. In order to integrate bound-checking we employ meta-interpreters, that bound specific predicates and ensure that their abstract syntax trees don't exceed a specific bound. For the randomness, we invoke our meta-interpreter to randomly choose which branch to explore but in our project, we were only able to implement the randomness to a limited extent: only at the leaf, and top level of the tree.
Figure 3 above lists the performance of our test case generator as a function of time. The cases are generated with bounds of 5 to ensure a well-balanced tree.
The results show that this approach is capable of generating test cases with parametric controls in the form of randomness, and bound checking. In practice, the parameters would have to be adjusted and extended to suit the software being tested. Example of parameters could range from implementing definitions of a specific set of grammar, with scoping for javascript; Hodovan et. al. implement a fuzzing approach that test DOM elements within javascript.

Conclusion

Using declarative definitions we were able to generate valid javascript features that could be utilized in a larger fuzzing scheme. Prolog's declarative nature, and its ability to reason about program execution(meta-interpretation) offers significant advantage by providing an intuitive way to generate test cases.
In the economics of software testing, resources are scarce, CLP offers an intuitive and optimized way to solve the problem of test case generation.

In our implementation, the parameters, and the variety of test cases were limited in terms of both complexity, and also in terms of practical feature testing. Some simple extensions would be to focus on a broader, and complete coverage of the javascript programming language; to cover a complex functionality of the javascript engine and how the engine handles it, for example, scoping, or inheritance.

In terms of lessons learned from working on this project, we would like to make three points:

1. CLP, especially prolog, has properties that can help solve

problems that require exhaustive search: problems that may take an indefinite time to solve for all cases but which for smaller cases, might have a solution.

2. Prolog is not a modern programming languages which makes simple i/o, or network operations difficult to implement.
3. Resources for learning prolog, and especially fuzz testing in prolog were limited compared to other programming languages, and their use..

References

[1] Dewey, Kyle et al.
Language Fuzzing Using Constraint Logic Programming. Copyright 2014 ACM 978-1-4503-3013-8/14/09

[2] Dewey, Kyle et al
Automated Data Structure Generation: Refuting Common Wisdom. Supported by NSF CCF-1319060.
ucsb.edu [viz]
https://sites.cs.ucsb.edu/~benh/research/papers/dewey15automated.pdf

**[3..] (various websites visited, eg in Chicago style (as opposed to APA or MLA) ; eg;
Lawrence et al. Dynamic and Functional Programming Topics. Published 2018. Web. Viewed Nov 22, 2019.**