Automatic program generation for detecting vulnerabilities and errors in JavaScript interpreters

Avital Shahar, Bar-David Itzik, Poleg Matan
Computer Science
Tel Aviv University
Spring 2014-15

Automatic program generation for detecting vulnerabilities and errors in JavaScript interpreters

Avital Shahar, Bar-David Itzik, Poleg Matan
Computer Science
Tel Aviv University
Spring 2014-15

Summary

In this workshop we have created an infrastructure for automatically generating javascript programs, running them on various javascript-engines and comparing their results.
The goal of the project was to find different behavior in some largely used javascript engines and report the bugs or unwanted behavior to their respective developers and thus getting them fixed.

In our work we have focused on non-visual features of the language (for example logical and mathematical calculations, functions, etc.) and found a few serious bugs and some unwanted behaviors mainly in dark corners of the javascript language.

Table of Contents

Summary i

Introduction 1

Work Method 1

Generating programs

Comparing output

Analyzing codes that provide different outputs on different engine

Results (main bugs) 2

Conclusions 3

Appendix A: Tested engines list 4

Appendix B: Generated code manipulation 4

Appendix C: python add-ons 4

References 5

Introduction

Javascript is a not-so-well-defined language, meaning a lot of its features were left to the interpretation of the engine's creator's decision, thus some things may be done differently in two engines and both of them are actually "correct". In order to overcome the uncertainty we took five largely used javascript engines[A] and defined an unwanted behavior as a behavior similar in four of the engines (The de-facto correct behavior) and different in the fifth.

Our work was inspired by csmith [1] so naturally the generated programs are assembled from: logical and mathematical calculations, variable and function definitions, assignments, and other non-visual features of the javascript language.

Work Method

Our work can be separated to three main threads:

1. Generating programs
2. Comparing outputs
3. Analyzing codes that provide different outputs on different engines

Generating programs

Our first step was creating an Abstract Syntax Tree (AST) that represents the javascript syntax each node of the tree represents a syntactic entity, after reviewing existing projects [4] [5] we decided to write from scratch. The second stage was using distribution formulas with configurable parameters to generate programs. At this point we have fine-tuned the configuration to generate more realistic programs. We also altered the generator to create programs that are somehow semi-randomized, with respect to every node's context (for examples see Appendix B)

During code analysis, as we encountered repeating issues that cause the engines to output different results, we added some injected code to the generated programs, to assure that the issues will not reproduce.

Note that every generated program begins with some pre written code that creates an API that allows us to overcome the differences we had already encountered and logged (for example since 'print' is called differently in some engines we created a proxy function that calls print with respect to the running engine).

Comparing outputs

In order to find different behavior of engines running the programs, we had injected code segments along the program to print the execution flow, and at the end of the program to print the values of all global variables.

On each run of jsfuzzer it generates a single javascript program (according to the input parameters[x]), executes it on each of the engines and compares the output (execution flow and global variables) bit-by-bit. It outputs which of the engines timed out, failed or succeeded and the actual output in equivalence classes (a single file for engines that had exactly the same output)

Since difference in executions became quite rare, in order to ease the process of finding code that yields different output on different engines we wrote a python script[C] that automatically runs jsfuzzer and saves generated programs with different output (with all files related to future debugging).

Analyzing codes that provide different outputs on different engines

Once we've generated a javascript file that yields different outputs on some engines we started a hands-on reverse engineering process to reduce the code to a minimal form that still causes difference. When we found major bugs we had reported them to their respective developers.

Results (Main bugs we had found)

The working process documentation is available in a wiki form at: https://jsfuzzer.wikispaces.com/
You are welcomed to use user: jsfuzz pass: jsfuzz

Dead code elimination - Nashorn

- Dead code elimination in a javascript file causes a runtime exception **in the engine**.
  In this case when a function contains a function definition in an unreachable area we presume that an optimization deleting contained functio n corrupts the containing one.
  http://bugs.java.com/ (JI-9023716)
- Dead code elimination in a javascript file causes a false failed run.
  In this version of the bug, dead code elimination causes corruption of the global scope.
  http://bugs.java.com/ (JI-9023808)

Function definition location

- DynJS – functions that are defined in conditional scopes (inside if statements) are undefined even though the execution got to the function definition location.
- NodeJS – functions that are defined in unreachable execution flows (not necessarily dead code) are known and runnable, as oppose to other engines in which the function is undefined

Small incompatibilities

- Rhino – when in a function $(undefined--)$ is treated as NaN in all engines but rhino, which evaluates it as undefined.
- Nashorn – the following expression: $(+(function()\{\}\ ||true))$
  is equivalent to NaN in all engines but nashorn, that evaluates it as 1

Cannot push undefined value to array – DynJS

When pushing undefined to an array, all engines but DynJS add a cell to the array with the value undefined, while DynJS pushes nothing (the array remains the same)
https://github.com/dynjs/dynjs/issues/157

Logical not error - DynJS

For every k $(k = 2^n\ s.t\ n \in \{32, \dots, 52\})$ the expression $!k$ yields true in DynJS as oppose to false in all other engines.
https://github.com/dynjs/dynjs/issues/156

JSON.stringify malfunction – DynJS

The expected value of $JSON.stringify(NaN)$ is null but DynJS returns NaN
https://github.com/dynjs/dynjs/issues/158

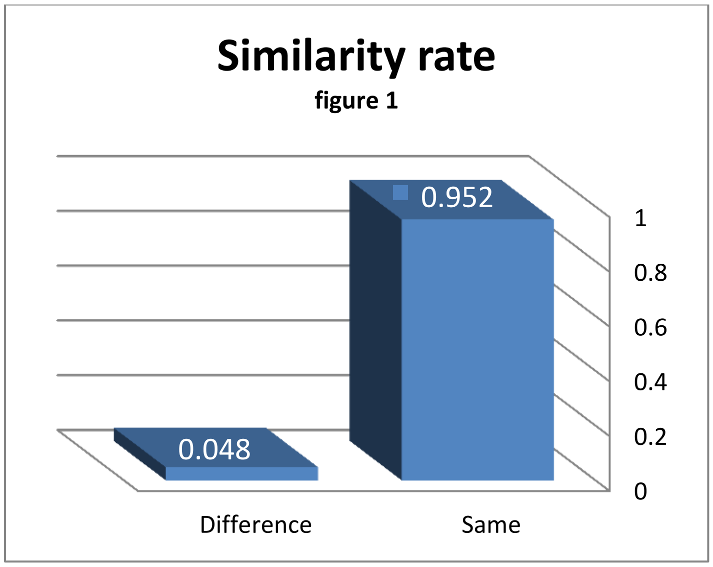

Figure 1 – As we can see in the visual presentation a vast majority of the executions outputted no difference (according to a late stage of the work)

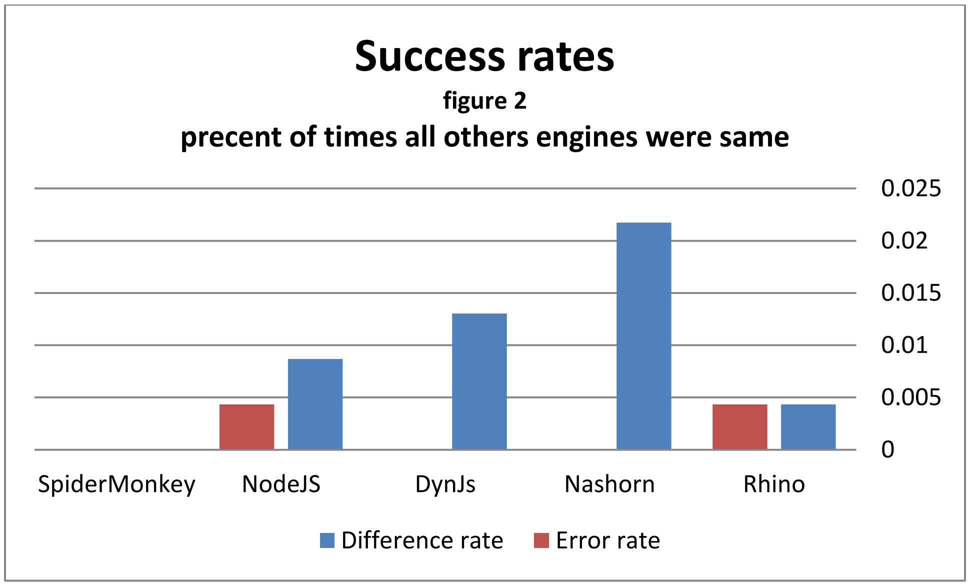

Figure 2 – Shows the relation of times each of the engines was the abnormal (with respect to the others), red shows failures when others succeeded and blue shows output different from the rest.

Conclusion

Jsfuzzer can function as a QA platform for javascript interpreters, one can easily generate random (or semi random using correct configuration) and compare the output of his engine of choice generates to some trustworthy engines.

We found a few interesting bugs that we believe are fix worthy, and some different approaches for javascript interpretation that are both in line with javascript standard.

Appendix A:
Tested engines list

We chose a variety of five largely used engines with consideration to how widely they are used in the world and ease of use:

- SpiderMonkey – Written in C++ by the *Mozilla Foundation* and used by Mozilla Firefox.
- Rhino – Developed in java by the *Mozilla Foundation* (no longer supported but still used)
- NodeJS – A wrapper for V8 (used by *chrome*) and very common in server side scripting
- Nashorn – First was developed in java by Oracle, was open sourced in 2012
- Dyn.JS – Open source by dyninc (rarely updated)

Appendix B:
Generated code manipulations

While generating the javascript code jsfuzzer injects code and forces some situations to achieve better programs:

- Global print function
- Handling with the halt problem
- Disabling object expression as a statement
- Wrapping call expressions, to avoid calling an undefined function
- Wrapping JSON.stringify function.
- Handling the member expression of null and undefined
- Avoiding Anonymous function as statement (not via a call)
- Disabling usage of ‘this’.

Appendix C:
Python add-ons

To ease the output analyzing we wrote some python scripts:

- $GetVarsOfCode.py$: Gets a code chunk and returns a var declaration line of all variable in the code chunk, This allows carving the code to reduce its size while maintaining the relevant parts
- $jsfuzz\_errorDetection.py$: Runs $jsfuzz$, and if there are interesting errors (such that we are unaware of from previous analyzing) save the code and output files and start over
- $jsfuzzRunner.py$: Runs $jsfuzz$ over and over, and saves the code and output files if two or more engines passed but got different output

References

1. Xuejun Yang, Yang Chen, Eric Eide, John Regehr, “Finding and Understanding Bugs in C Compilers” *University of Utah, School of Computing.*

2. Robert Sedgewick, Kevin Wayne, “StdRandom.java”

3. A tutorial for javascript, *w3schools.com/js/*

4. *http://jointjs.com/demos/javascript-ast*

5. *http://lisperator.net/uglifyjs/ast*