FaJITa: Verifying Optimizations on Just-In-Time Programs

David Thien
UC San Diego
dthien@eng.ucsd.edu

Evan Johnson
UC San Diego
e5johnso@eng.ucsd.edu

Michael Smith
UC San Diego
mds009@eng.ucsd.edu

Sorin Lerner
UC San Diego
lerner@cs.ucsd.edu

Fraser Brown
CMU
fraserb@andrew.cmu.edu

Hovav Shacham
UT Austin
hovav@cs.utexas.edu

Deian Stefan
UC San Diego
deian@cs.ucsd.edu

Abstract

We introduce FaJITa: a translation validation tool for verifying semantic equivalence after optimizations. FaJITa combines symbolic execution with more traditional static analysis techniques to verify functions which have passed through Firefox's optimizing JIT compiler. We are able to verify practically all JavaScript functions, without having to resort to dynamic techniques or branch pruning to decrease the state space. We additionally develop a fuzzing infrastructure to look for bugs in the Firefox JIT, as well as debugging tools for distinguishing true bugs from false positives.

Keywords translation validation, JIT, static analysis

1 Introduction

Modern web browsers are among the most widely-used and security-critical pieces of software today. Browsers must accept arbitrary JavaScript code from untrusted sources, then compile and run it in a safe way, all while maintaining competitive performance in the rapidly-evolving race between the major vendors. The JavaScript engines which power these browsers are one of the main sources of complexity, involving several different interpreters and compilers which make tradeoffs between startup time and execution time of the code. A bug in any part of this engine can lead to an exploit [3] compromising the user's system. Despite the significant efforts from vendors to design secure browsers, new bugs are still regularly found [1, 4].

One of the common places in the engine that exploit bugs are found is optimization passes[5]. A bug in an optimization pass can remove an array bounds check, or try to access fields of an object which no longer exist, resulting in an out of bounds read or write. Although there has been significant success using fuzzing to find bugs [2], these efforts have focused on coverage of the entire engine. This casts a very wide net for testing possible sources of bugs, but doesn't look over any particular part in very fine detail. To this end, we develop FaJITa, a translation validation tool which can verify optimizations have been correctly performed for particular programs. This tool, in conjunction with targeted fuzzing techniques, allows us to target a specific and dangerous part of the engine, very precisely.

2 Design

FaJITa's translation validation operates on Firefox's MIR (Middle-level Intermediate Representation). MIR is a language which mixes the high-level operations of JavaScript programs with lower-level engine implementation details. MIR programs represent JavaScript functions, and are IR most optimizations are performed.

FaJITa accepts as input 2 MIR programs which represent the same JavaScript function before and after any number of optimizations. It will then analyze the programs to ensure all observable effects are the same in both versions. If FaJITa's analysis decides the two versions will always have the same behavior, it declares them equivalent.

At its core, FaJITa is a symbolic execution tool. Thus, its analysis operates on all possible inputs to the functions, and all possible memory states. It will then symbolically execute the program, and ensure the following effect are equivalent in all possible executions:

- input object mutations
- output values
- global object mutations
- state updates (e.g. engine calls)

Critically, FaJITa is able to verify nearly all JavaScript-compiled MIR programs without having to mix dynamic analysis, or add limitations to control flow constructs. This is thanks to several analysis simplifications we were able to make.

2.1 Analysis

Only model a subset of the MIR semantics. Traditionally, if symbolic execution tools want to be fully-precise in their analysis, they must completely model the semantics

of their target language. However, we don't care about the exact behavior of any single program; we only care about the *difference* in behavior between 2 programs. Thus, we can approximate almost all MIR operations as uninterpreted functions, and only fully implement the subset of the semantics the optimizations can reason about.

For example, suppose the compiler decides to move a variable declaration out of a loop. In this case, we don't care the specific semantics of the operation, so long as the same operation is performed in both versions of the program. Note that this isn't true for side effecting operations (where the order operations are performed matters), so we must add additional semantics for those. This code motion optimization is in contrast to something like constant folding, where — in order to verify the optimization is correct — we must specify the full semantics for arithmetic operations.

Analyze loops independently. One of the traditional sticking points with symbolic execution is unbounded loops. These are difficult for tools to reason about because each time a loop executes, another possible symbol branch is added to the execution trace. We again get to make a nice simplification here, because we only want to understand when two programs might have divergent behavior.

Instead of modelling all possible branches in a loop, we only check that the behavior of both loops is the same, as well as that the branch conditions are equivalent. We recursively check nested loops, and allow for loop-invariant code-motion thanks to an additional analysis we omit here.

Mix symbolic execution with traditional static analysis. Even with these simplifications, doing full symbolic execution on MIR programs proved infeasible due to their size, and the complex memory model used in JavaScript engines.

An object in Firefox is primarily comprised of its slots (the list of values the object contains), and its shape (the mapping from field names to indices in the object's slots). This layout adds several layers of symbolic indirection between looking up a field of an object, and getting the corresponding value, and slowed down the symbolic execution engine's underlying SMT solver to a standstill.

Rather than try to do this analysis entirely symbolically though, we were able to take advantage of the structure of MIR programs to track an object's shape statically, that is, in a separate analysis pass instead of in the SMT solver. This optimization removed a layer of symbolic indirection from the engine, and made verification feasible. There were several other parts of the analysis we tracked statically rather than symbolically, including several instances of type information, global variable locations, live objects, and function calls.

2.2 Engine complications

Our analysis was further complicated by several complications created by JavaScript and the JIT. One of the key features of modern JavaScript JITs that makes them so fast, is that programs can have speculative optimizations performed on them. Speculative optimizations are optimizations done under certain assumptions about the types or values of parameters to future calls of a function.

For instance, if the compiler notices that a function is always called with a string as input, it can specialize the operations in that function to always be string operations, rather than having to look up the more general version every time. This comes with a caveat: functions that have had speculative optimizations performed on them must dynamically check to ensure the assumptions they made still hold. If the compiler performs this string optimization, but forgets to check that the parameter is a string every time, suddenly the function can be called with a non-string parameter, and the behavior can change. To address this, FaJITa verifies the *subset* of inputs to the program that the program is specialized for, but *only* if the check exists. If the check is missing, FaJITa will notice that the behavior diverges, and properly mark the 2 MIR programs as not equivalent.

There were additional complications imposed by the engine thanks to features like garbage collection, the details of which we omit here.

3 Evaluation

We see FaJITa as being useful in two ways:

- as a verification tool for verifying optimizations have been performed correctly
- as a bug-finding tool to be used in conjunction with fuzzers.

In both cases, the tool is most useful with a low false positive and false negative rate. To assess the false positive rate, we ran FaJITa on a combination of fuzzing programs, hand-written tests, and Firefox's JavaScript test suite. Across more than 150,000 JavaScript programs, we have about a 0.01% false positive rate. It is more difficult to assess the false negative rate, because there is no ground truth. To check this as best we could, we hand-crafted pairs of MIR programs that had different behavior in subtle ways, and recreated previous bugs in optimizations passes, all of which FaJITa is able to catch.

For verification, FaJITa takes advantage of a framework we previously developed, which allows it to get access to MIR programs while the engine is running. We can then run verification on a separate thread whenever the engine doesn't need the resources.

For bug-finding, we are currently running FaJITa in conjunction with Fuzzilli [2] to generate JavaScript programs. We are additionally experimenting with using heuristic gleaned from running FaJITa to guide Fuzzilli toward JavaScript programs we believe will be more likely to expose bugs in the engine.

References

[1] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. Jit-Picking: Differential Fuzzing of JavaScript Engines. In *CCS*.

[2] Samuel Groß. 2019. Fuzzilli. https://github.com/googleprojectzero/fuzzilli.

[3] Samuel Groß. 2020. JITSploitation I: A JIT Bug. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html.

[4] Mozilla. 2022. Mozilla Foundation Security Advisory 2022-40. https://www.mozilla.org/en-US/security/advisories/mfsa2022-40/.

[5] Tobias Tebbi. 2022. Chromium Gerrit: validate more concurrent reads. https://chromium-review.googlesource.com/c/v8/v8/+/3936273.