CS 6340 Review: Towards a verified range analysis for JavaScript JITs

Diptark Bose (GT ID: 903613944)
August 29, 2021

1. Summary

Background

Browser JITs compile and execute all the JavaScript present in web pages. We all want our web-browsers to be fast. And to make it speedy, developers come up with ways to make the Browser JIT run faster. One such optimization that is employed is ***Range Analysis***.

Compilers use range analysis to deduce ranges (i.e., the upper and lower bounds) of values at different points in program execution. Once these ranges have been deduced, the compiler then uses these results to inform its optimization passes to generate efficient code. The paper mentions two such optimizations - Dead Code Elimination (DCE) and Bounds Check Elimination (BCE).

Since JavaScript is a memory-safe language, accessing a memory location entails bounds check, wherein the JIT confirms whether the attempted access is within bounds. If the address being accessed is out of bounds, ***undefined*** should be returned. These repeated bounds checks cost time, and developers need a way to minimize this. Therefore, the next natural step for Browser developers is to write range analysis routines that could be used to check indices, and if it's within bounds, inform the JIT that further bounds checks aren't necessary. This is known as ***Bounds Check Elimination (BCE)***, and it results in a faster compilation of JavaScript objects, thus leading to fast web page rendering.

What problem are the authors trying to solve?

Understandably, such optimizations come with their own set of risks. From the above explanation of range analysis, one can easily deduce that ***if the range analysis is wrong, it can allow attackers to craft JavaScript code to access out-of-bounds memory***.

Let's talk about one possible way through which bounds checks are erroneously eliminated.

JITs are engineered to optimize array accesses when all elements are of the same type (i.e., the array shape is stable). However, if the array is filled with objects of different data types, the JIT should ideally switch to it's slower, unoptimized path for the sake of accuracy. When the JIT fails to switch, it leads to type confusion, which incorrectly eliminates the bounds check. This results in out-of-bounds memory access, which is a glaring gap that invites cyber attacks. In fact, this vulnerability has been exploited earlier as well. Thus, to quote the authors, ***unverified range analysis can be "weaponized against users"***.
Therefore, there is a need for a tool that could *verify* the range analyses routines of Browsers.

What is novel in this paper? What challenges did the authors overcome?

To address such problems, the authors propose a state-of-the-art, novel compiler verification process to secure Browser JIT compilers. This layer of verification can confirm compiler sanity by applying formal methods that consider all possible edge cases. ***The authors present VeRA, a tool that can do this sanity verification on the range analysis routines in Browser JITs***.

VeRA is a subset of C++ for writing verified range analyses routines. VeRA also supports a Satisfiability Modulo Theories (SMT) based verification tool that proves sanity of analyses with respect to the JavaScript semantics. Developers can now use VeRA to write their analysis logic and can directly compile it for custom properties' correctness proofs.

Speaking about challenges, this one instance comes to mind. VeRA C++ needs to be translated into the SMT, and this was a challenging task for two main reasons:

1. SMT solvers usually define low-level operators like logical bit shifts and assignments. They don't provide support for higher level C++ constructs like conditional branches and functions.
2. The semantics of SMT types are different from their respective C++ counterparts.

In yet another novel pursuit, the authors tackled this in a very ingenious way, which I found to be quite interesting, and have highlighted in the next section.

What I found interesting?

In order to tackle the incompatibility with SMT, the authors came up with the idea of developing an Intermediate representation (IR) between C++ and SMT. So the compiler first translates C++ to IR. This step takes care of rewriting control flow constructs like branches, function calls, and return values. Once this step is complete, a translation from IR to SMT takes place, which bridges the gap between the semantics of the languages.

1. **Compiling C++ to IR**
   This compilation is a series of transformations that needs to translate higher level constructs like branches and function calls to acceptable lower level constructs. If statements, function calls, and method calls need to be mapped to corresponding SSA assignments. This is essential as the IR is expecting SSA assignments.
2. **Compiling IR to SMT**
   The previously generated IR is next translated to SMT. The assignment nodes are translated to equality, following which the expression tree in RHS is traversed in a bottom-up manner for converting to SMT.

So finally, the ***authors were able to convert C++ to SMT by introducing their own IR*.** Readers might wonder why the authors couldn't use an existing IR for this translation. The authors explain this by mentioning that no LLVM-IR-level tool supports JavaScript semantics, and some don't even support C++ reliably. Thus, the authors had to take this route for a successful translation of VeRA C++ to SMT.

Results

In this section, I list out all the results mentioned in the paper:

1. Authors confirm that the ***range analysis routines verified by VeRA pass all 140,000 Firefox JIT Tests***, while showing very minimal changes in the median page load latencies.
2. VeRA's verification ***uncovered a new Firefox range analysis bug*** and additionally confirmed an old bug from a previous version of the browser.
3. Authors verify 21 range analysis routines, out of which ***86% were either proved correct or refuted***.
4. Verification of the range analyses for ***union and intersect functions timed out***.
5. Range Analysis verification using VeRA C++ showed ***no performance regressions***.

Future Scope

The authors have hinted about some possible avenues which can be improved in future endeavors. Thus, these tasks can be described as the future scope:

1. VeRA C++, being a subset of C++ and currently not supporting various standard C++ features like loops and recursion. Adding support for these constructs would make VeRA more powerful.
2. LLVM-IR doesn't have JavaScript support. This could be a very useful utility if developed.

2. An Illustrative Example

In this section, I aim to reproduce the bug that the authors were able to find using VeRA, and write simple code for exploiting this bug. I hope my example demonstrates the dangers such a small bug could pose. This is a bug with the ***ceil function of Firefox***, which is used to round up the number to the next largest integer.

Firefox's range analysis routine keeps track of ranges using variables for upper and lower bounds of ranges. In addition to these two variables, the routine also houses metadata, such as ***canHaveFractionalPart*** (which indicates if the range is allowed to be composed of fractional numbers), ***canBeNegativeZero*** (which indicates if the range can have starting or ending value as -0), and more such useful flags.

Now, let's talk about the working of ceil function in Firefox. The ceil bumps the number up to the ***next largest integer***, and while doing so, updates the canHaveFractionalPart to false (i.e, not possible), since an integer is not supposed to have decimal parts. But what the Firefox developers failed to anticipate is the case when we run something like ceil(-0.5), which returns the answer as a negative zero. ***The ceil routine isn't programmed to set the canBeNegativeZero flag to true for accommodating this possibility***.

Now, here's a code snippet that I wrote for exploiting the bug.

```
let arr = [1, 2, 3, 4, 5, 6];
let index;
if(Math.ceil(-0.5) === -0)
  index = true;
else
  index = false;

console.log(arr[index*4000]);

```

The idea here is to fool the JIT into believing that the accessed index is always in bounds. In the above code, in line 3, since ***canBeNegativeZero*** is never set as true by the ceil function, the condition believes that it can never be -0, thereby ***declaring the condition as always false***. This confuses the JIT, as ***it starts believing that the 'index' would always have the value of false***. This would mean that the accessed index in line 8 would always be 0. So the JIT is misled to think that the accessed index in line 8 will always be 0, which is in-bounds, and it eliminates all future bounds checks.

When JIT does this erroneous BCE, line 8 becomes exploitable, as it let's us access index 4000, which is clearly out-of-bounds.

This Firefox bug has been unearthed now, thanks to VeRA!

3. Questions

- What is the reason behind differences in page load latencies in Figure 9? I observed that for some websites, the VeRA integrated page load latencies were more than regular, while for some it reduced. Shouldn't page load latency increase/decrease with VeRA for all websites in a similar manner?
- Why weren't other static analysis routines investigated using VeRA? Does VeRA lack the support for that, or were other analyses simply out of scope for this paper?