Validating the JavaScript Rosetta Stone

Chase Killorin and Deepak Kurian

12/3/2025

Abstract

Our work consisted of improving upon our baseline Turbo-TV, a translation validator for just-in-time compilation (JIT) in V8. Many bugs in modern browsers arise due to the optimizing JIT compilation; our primary target is Turbofan within V8. Turbofan makes assumptions during its optimizing JIT passes that are not always consistent with the source code. These discrepancies can, for example, remove a “redundant” bounds check. This being missed can lead to a powerful primitive that can result in a vulnerability in V8. These inconsistencies can be checked through translation validation, and Turbo-TV accomplishes this with its equivalence and undefined behavior checks. There yet lies a few limitations in the completeness of the equivalency check. There is room for additional opcode support for unknown and unsupported opcodes, such as SpeculativeSmallIntAdd and JSStoreGlobal. Our contribution was encoding the semantics for these operators within Turbo-TV and giving the equivalency check the ability to validate optimization translations with these opcodes. Support or efficacy for these was not previously supported before, as our work enabled reasoning for these opcodes within tested JavaScript programs.

Introduction & Background

Chrome has a market share of around 71%. Additionally, more than 5% share from other browsers are Chromium-based [5], which means they rely on the same underlying code that would also make Chrome vulnerable. Within Chromium lies V8, the JavaScript engine, from which many memory corruption vulnerabilities leading to remote code execution (RCE) come from. A major source of these memory corruption issues resides in the inaccurate optimized JIT compilation that the underlying JIT compilers make, e.g, Turbofan. The research problem exists to improve V8 JIT security by improving upon existing tools, in our scenario, Turbo-TV. The objective we contributed to was supporting additional opcodes within Turbo-TV.

First, the difference between ahead-of-time (AOT) and just-in-time compilation. The standard form of compilation you see for a typical C/C++ program is AOT. The flow occurs of your source code being preprocessed and compiled by a tool, such as gcc or Clang, into object files. These object files are then linked against shared libraries, either statically or dynamically, and your final executable results from this. In something such as Java or JavaScript, some, if not all, compilation is done within a JIT compiler. The JIT compilation tracks code that is often run, considered “hot”, then lowers (further compiles) and, importantly for translation validation, optimizes the code further.

This ties into V8 with Turbofan, the optimizing compiler in the multi-tiered engine that is V8. Turbofan, or how it has changed now, Turboshaft, is the main target for the translation discrepancies when JavaScript code is being compiled. It contains the optimization passes that leave room for the compiler to make sharp changes that can alter a simple redundancy check.

One small oversight in how memory is handled in the memory-unsafe C++ that Turbofan is coded in can cause RCE.

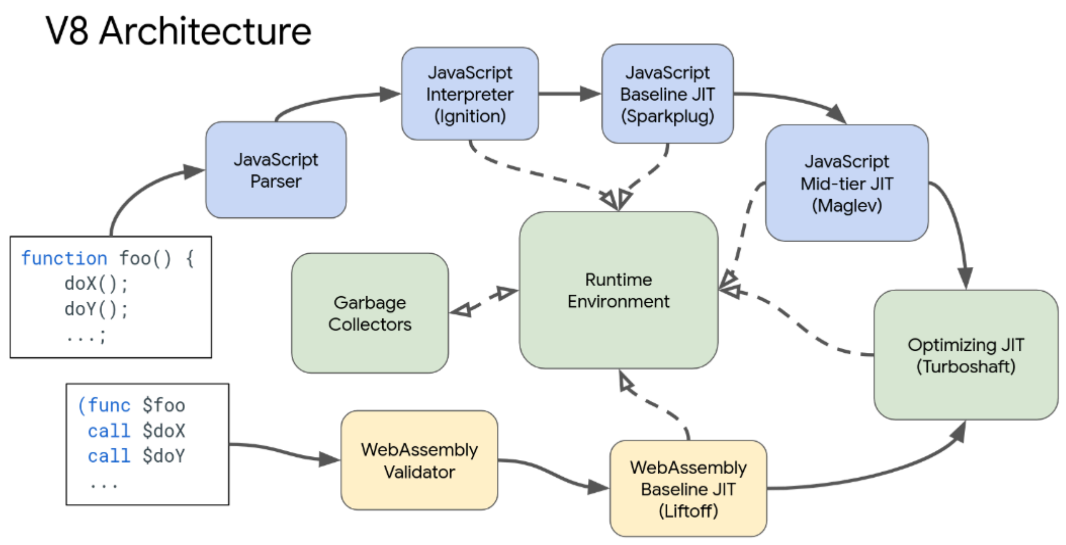

V8 Architecture
JavaScript Parser
JavaScript Interpreter (Ignition)
JavaScript Baseline JIT (Sparkplug)
JavaScript Mid-tier JIT (Maglev)
Optimizing JIT (Turboshaft)
Runtime Environment
Garbage Collectors
WebAssembly Validator
WebAssembly Baseline JIT (Liftoff)
function foo() {
doX();
doY();
...;
(func $foo
call $doX
call $doY
...

Figure 1, [6]

Translation Validation, or the TV in Turbo-TV, takes two versions of the compiled code. The optimized and the unoptimized versions, these versions of the code, the src and tg,t are both in Turbofan IR, represented in the form of a Sea-of-Nodes (SoN). This is the state these programs are reasoned at, an intermediary representation that allows us to dispute semantic equivalency. Let's take a simple program performing arithmetic. This program has two functions, with one that can be inlined within the other. The program shown below is functionally identical to the IR shown in *Figure 2*.

```
JavaScript
function b(x) { return x+2;}
function a(x) {      y = x+b(x);

                     reutrn y;     }
```

Now we can analyze the pre-optimized version on the left, BytecodeGraphBuilder, and on the optimized version on the right, EarlyOptimization. The semantic equivalency between these two is fine in this case, but it is the two versions of the programs' IR that Turbo-TV is able to check. The left side in *Figure 2* contains the JSStoreGlobal and JSLoadGlobal; this is due to the separate function being called and needing to move its context. On the right side, we can now see that the program has inlined the store and load; now we simply make a call and invoke the addition. Introducing these semantics for JSStoreGlobal was one of our contributions.

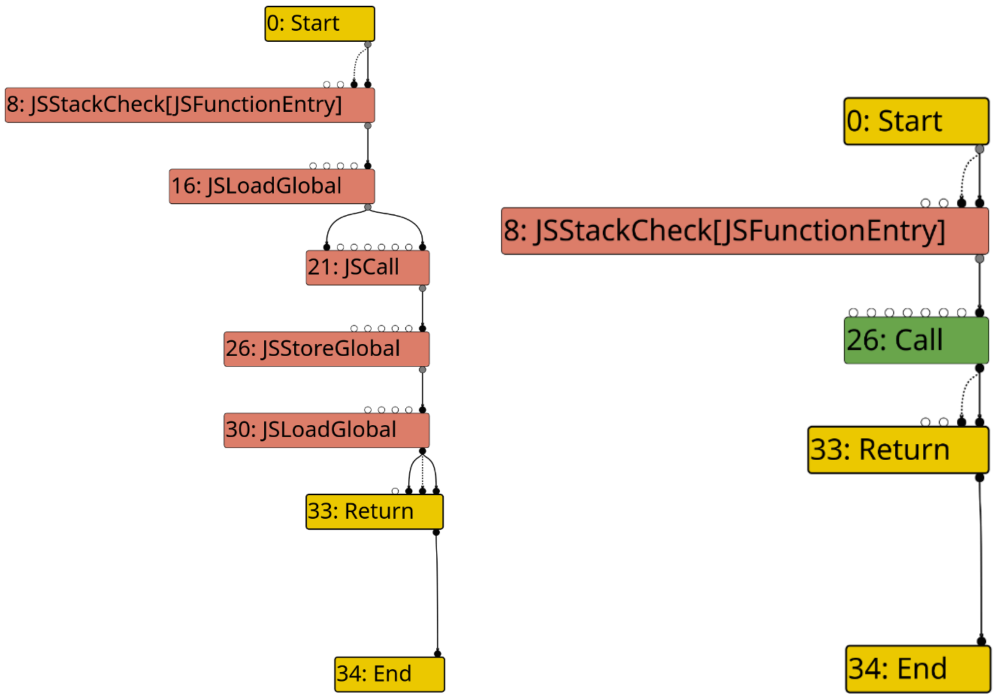


Figure 2

Related Work

Most related work on Translation Validation (TV) is on AOT compilation. Fewer works discuss TV on JIT compilation. In the case of V8, not only does V8 use a JIT, but it also uses Sea-of-Nodes, which is a graph representation used by V8 to represent its Intermediate Representation.

VeRA [3] implements a formally verifiable Range Analysis validator, which is built into Firefox's JIT compiler. This ensures that any translation that uses the Range Analysis pass will always be correct and won't cause any bugs/vulnerabilities. The main downside of this technique is that it is time and resource-intensive to formally verify the internals of a JIT compiler and implement this proof inside the compiler. Additionally, the verification code will need to be rewritten every time the Range Analysis code is modified. VeRA also only supports a subset of C++, which may affect the developer's ability to add features. This is the main trade-off that VeRA considers.

Alive2 [2] is a Translation Validation tool for LLVM's IR. It checks for semantic equivalency between the source and target IR (before and after passing through LLVM's optimization passes) to find bugs in LLVM's JIT. One of the issues the authors faced was the fact that LLVM's IR had no formal semantic definitions. As a result, each opcode had to be analyzed to understand its semantics and then implemented. Alive2 was implemented as a standalone tool that can analyse programs outside of the LLVM compiler. This has the added benefit of being compatible with multiple versions of LLVM without any modification to Alive2. It also had no performance drawbacks, but unlike VeRA, it is unable to identify all bugs in the JIT compiler (or, in the case of VeRA, the part of the compiler that VeRA verifies).

Baseline

The baseline used for our project is TurboTV [1]. TurboTV performs translation validation on V8's JavaScript JIT. It traverses the SoN graph and interprets each Opcode while maintaining its own version of V8's internal state. When TurboTV runs into an Opcode it understands, it modifies the state appropriately and starts to build an SMT equation to represent that Opcode. This process continues until TurboTV reaches the end of the graph. It repeats this step with the source and target IR until it creates an SMT equation for both, then compares the two equations to check for semantic equivalency. This is similar to Alive2 [2] but works on V8's IR instead of LLVM's IR. TurboTV is the only paper to apply Translation Validation on v8 to identify semantic vulnerabilities.

A high-level overview of how TurboTV's workflow is shown in *Figure 4.*
Given a JavaScript file, the first step is to extract its IR, both pre and post-optimization. This is done using a build of v8. These IR files are what are passed into TurboTV as input. For the Undefined Behaviour Checker, it analyses each IR file individually to find bugs. For the equivalency checker, TurboTV compares 2 IR files to make sure they are semantically equivalent. The 2 files are the pre and post-optimization IR files of the same input JavaScript source code. Once you create a corpus of JS files and their corresponding IR files, you can run TurboTV against them to find bugs ahead of time.

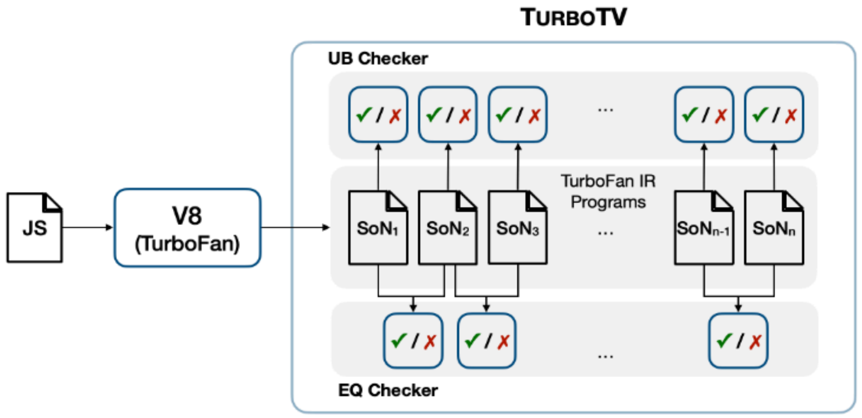


Figure 4

TurboTV uses semantic definitions for the Sea of Nodes defined in this paper [5]. Despite having semantic information about SoN, there are no formal semantic definitions for v8's Opcodes themselves. This makes it challenging to define new Opcodes in TurboTV, as you need to understand how the Opcode is implemented in v8 to create an equivalent definition in TurboTV in order to create an accurate SMT equation.

The internal state representation used by TurboTV is shown in *Figure 3*. The State represented by a tuple has the following values.

- Label: The ID of the current node. It can be thought of as a program counter register.
- Registry File: A mapping from registers to values.
- Memory: Representation of internal Memory used to store objects.
- Deoptimization Flag: A flag indicating whether deoptimization has been triggered.
- Undefined Behaviour Flag: A flag indicating whether Undefined Behaviour has occurred.

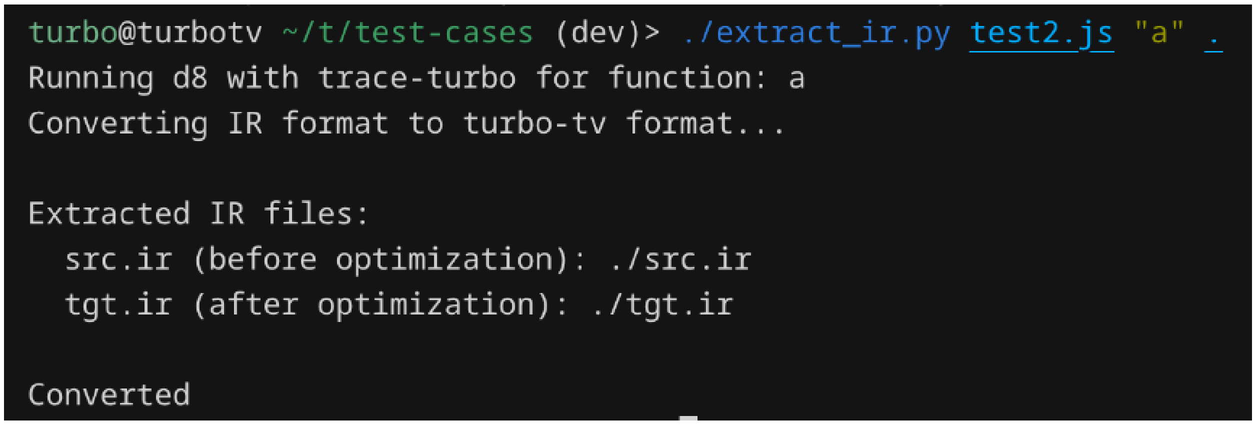
$$\begin{array}{rcl}
\mathit{State} & = & \mathit{Label} \times \mathit{RegFile} \times \mathit{Memory} \times \mathit{Deopt} \times \mathit{UB} \\
\mathit{RegFile} & = & \mathit{Register} \rightarrow \mathit{Value} \\
\mathit{JSValue} & = & \mathit{TaggedPointer} \uplus \mathit{TaggedSigned} \\
\mathit{Value} & = & \mathit{JSValue} \uplus \mathit{Bool} \uplus \mathit{Int8} \uplus \mathit{Int16} \uplus \mathit{Int32} \uplus \cdots \\
\mathit{TaggedPointer} & = & \mathit{BlockID} \times \mathit{Int32} \\
\mathit{TaggedSigned} & = & \mathit{Int31} \\
\mathit{Memory} & = & \mathit{BlockID} \rightarrow \mathit{Block} \\
\mathit{Block} & = & \mathit{Byte}^{*}
\end{array}$$

Figure 3

When TurboTV comes across an Opcode that it does not understand/implement, it stops calculating the SMT and returns an error message. This is important because TurboTV does not implement certain key functionality, such as “Function Calling” (Opcode: **JSCall**), “Loop Unrolling” (Opcode: **Loop**), and does not support various opcodes.

Methodology

Our methodology consists of creating simple JavaScript programs, which only need to produce unsupported opcodes. We translate the program to its src and tgt IR, identify TurboTV’s support, and add semantic knowledge to the definitions of the operators. If that relieves the issue, the program is altered, and we repeat. To start, we first need to lower our JavaScript test cases to IR using d8, the V8 debugging tool. This gives us the src and tgt IR files that we can pass to Turbo-TV’s equivalency check so it can reason.



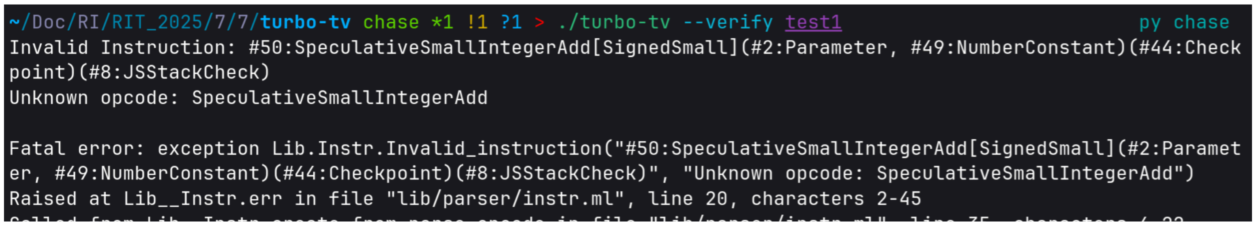
```
turbo@turbotv ~/t/test-cases (dev)> ./extract_ir.py test2.js "a" .
Running d8 with trace-turbo for function: a
Converting IR format to turbo-tv format...

Extracted IR files:
  src.ir (before optimization): ./src.ir
  tgt.ir (after optimization): ./tgt.ir

Converted
```

Figure 4

We then are able to run the equivalency checker on the IR (Turbo-TV check JS files directly in production). Here we can see that we have an unknown opcode of SpeculativeSmallIntegerAdd. This opcode is simply what V8 uses to perform a speculative smi add. This error message from inside **instr.ml** tells us that the semantics for this opcode do not exist within TurboTV.

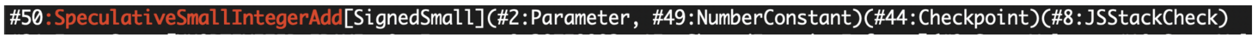


```
~/Doc/RI/RIT_2025/7/7/turbo-tv chase *1 !1 ?1 > ./turbo-tv --verify test1                  py chase
Invalid Instruction: #50:SpeculativeSmallIntegerAdd[SignedSmall](#2:Parameter, #49:NumberConstant)(#44:Check
point)(#8:JSStackCheck)
Unknown opcode: SpeculativeSmallIntegerAdd

Fatal error: exception Lib.Instr.Invalid_instruction("#50:SpeculativeSmallIntegerAdd[SignedSmall](#2:Paramet
er, #49:NumberConstant)(#44:Checkpoint)(#8:JSStackCheck)", "Unknown opcode: SpeculativeSmallIntegerAdd")
Raised at Lib__Instr.err in file "lib/parser/instr.ml", line 20, characters 2-45
```

Figure 5

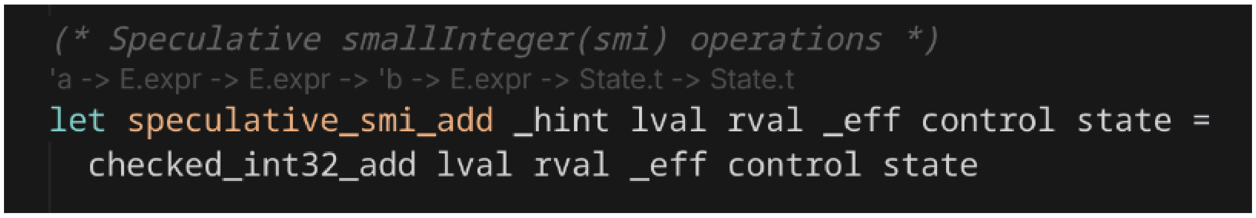
We can view the line within the IR that is giving TurboTV issues, which also provides an instance of the semantics. We can read the hint of what the value is expected to be, SignedSmall. Then the left value, Parameter, and the right value, NumberConstant. Lastly, we have the effect and control nodes that are required before SpeculativeSmallIntegerAdd can run. This is where SoN comes in and can reorder instructions based on these constraints for optimizations.



```
#50:SpeculativeSmallIntegerAdd[SignedSmall](#2:Parameter, #49:NumberConstant)(#44:Checkpoint)(#8:JSStackCheck)
```

Figure 6

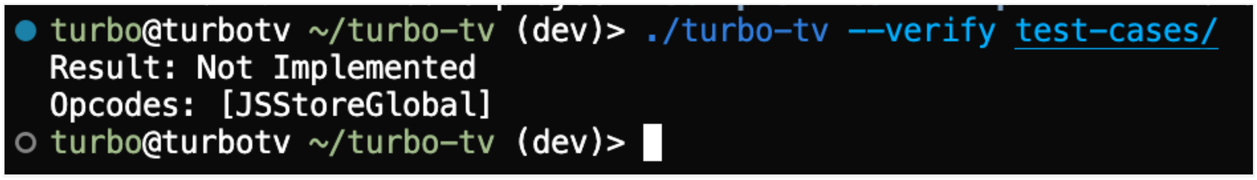
We define the semantics and can assume the speculative_smi_add is going to be, at the end of its state, equivalent to a basic checked_int32_add. This lets us give TurboTV a way to understand SpeculativeSmallIntegerAdd and validate it.



```
(* Speculative smallInteger(smi) operations *)
'a -> E.expr -> E.expr -> 'b -> E.expr -> State.t -> State.t
let speculative_smi_add _hint lval rval _eff control state =
  checked_int32_add lval rval _eff control state
```

Figure 7

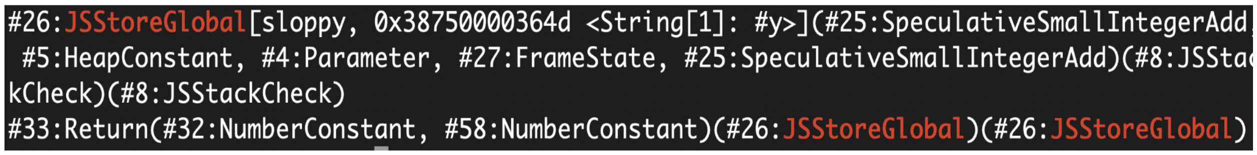
After SpeculativeSmallIntegerAdd, we rebuild and run our verify case again. Now we see that TurboTV complains about an unimplemented opcode, JSStoreGlobal. This opcode is the basic JS call for storing a global variable into memory. This is exactly the type of state that the equivalency checker in TurboTV tracks. This is compared in the src to the tgt IR, so any inconsistencies with this global variable existing would be something TurboTV identifies.



```
● turbo@turbotv ~/turbo-tv (dev)> ./turbo-tv --verify test-cases/
  Result: Not Implemented
  Opcodes: [JSStoreGlobal]
○ turbo@turbotv ~/turbo-tv (dev)>
```

Figure 8

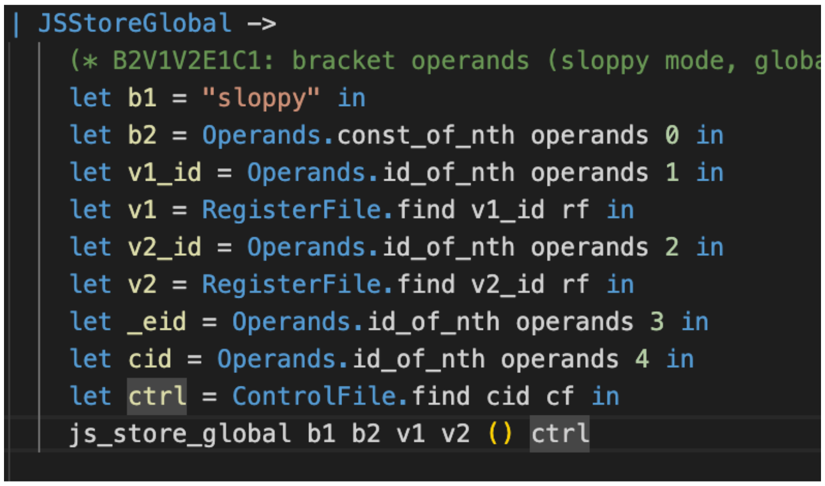
The semantics here are slightly more complicated here (*Figure 9*). There are 2 hints for this instruction. The first indicates that the program is running in Sloppy mode as opposed to Strict Mode [8]. The second hint is the variable name that is to be stored. While the variable name is relevant to developers, it isn't as important to the internal state, as we are more concerned about the location/memory address at which the value is to be stored. In our example, the value is node 25: SpeculativeSmallIntegerAdd, and the address at which it is to be stored is node 5: HeapConstant. This means that the result of node 25 is to be stored at the memory address stored in node 5.



```
#26:JSStoreGlobal[sloppy, 0x38750000364d <String[1]: #y>](#25:SpeculativeSmallIntegerAdd
 #5:HeapConstant, #4:Parameter, #27:FrameState, #25:SpeculativeSmallIntegerAdd)(#8:JSSta
kCheck)(#8:JSStackCheck)
#33:Return(#32:NumberConstant, #58:NumberConstant)(#26:JSStoreGlobal)(#26:JSStoreGlobal)
```

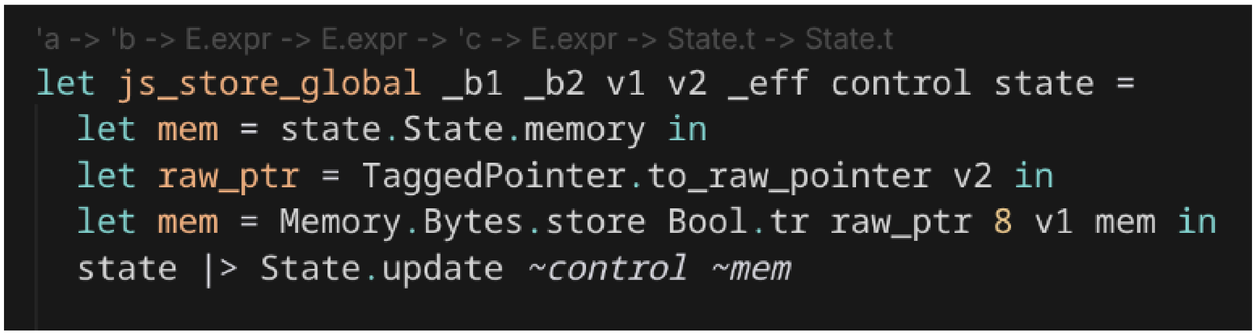
Figure 9

Our fixes consisted of adding parser logic for the JSStoreGlobal Opcode. By looking at the IR in *Figure 9,* we can understand the arguments for the opcode. The changes we implemented are shown in *Figure 10.* The only values required to implement the opcode are v1 and v2, which contain the value of the variable and the heap address, respectively. These values are passed into a different function, **js_store_global()** (*Figure 11*). This function retrieves the global state variable and accesses the memory variable **mem**, identifies the raw memory address from the heap node, and stores the value at that memory location in **mem.** It then updates the state with the new memory and control so that subsequent operations can access this memory value and create the SMT logic for JSStoreGlobal.



```
| JSStoreGlobal ->
    (* B2V1V2E1C1: bracket operands (sloppy mode, globa
    let b1 = "sloppy" in
    let b2 = Operands.const_of_nth operands 0 in
    let v1_id = Operands.id_of_nth operands 1 in
    let v1 = RegisterFile.find v1_id rf in
    let v2_id = Operands.id_of_nth operands 2 in
    let v2 = RegisterFile.find v2_id rf in
    let _eid = Operands.id_of_nth operands 3 in
    let cid = Operands.id_of_nth operands 4 in
    let ctrl = ControlFile.find cid cf in
    js_store_global b1 b2 v1 v2 () ctrl
```

Figure 10



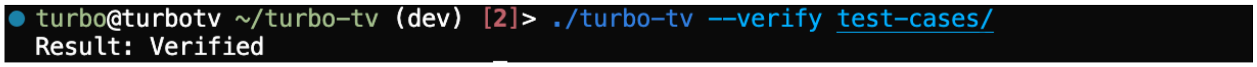
```
'a -> 'b -> E.expr -> E.expr -> 'c -> E.expr -> State.t -> State.t
let js_store_global _b1 _b2 v1 v2 _eff control state =
  let mem = state.State.memory in
  let raw_ptr = TaggedPointer.to_raw_pointer v2 in
  let mem = Memory.Bytes.store Bool.tr raw_ptr 8 v1 mem in
  state |> State.update ~control ~mem
```

Figure 11

Results

Due to the timeframe of the project, we decided to focus on a few instructions/test cases rather than creating a corpus using a tool such as Fuzzilli and work incrementally to implement more opcodes as we proceed. Since our contributions involve implementing new Opcodes, our results only show TurboTV analysing JS programs with these opcodes instead of throwing an error as it used to previously.

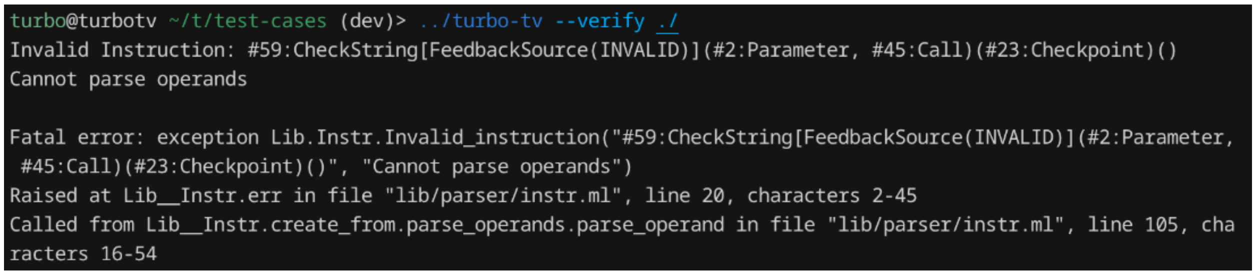
We added support for JSStoreGlobal and SpeculativeSmallIntegerAdd, allowing us to verify the previous testcase (*Figure 12*). This also provides valuable insight into the semantics of these Opcodes, which helps us implement other similar Opcodes such as JSLoadGlobal (whose logic would be the same as JSStoreGlobal, but reading from the memory address instead of writing to it) and other Speculative Small Integer Arithmetic operations such as subtract, multiply and divide (which would be redirected to its Checked Integer equivalent).



```
● turbo@turbotv ~/turbo-tv (dev) [2]> ./turbo-tv --verify test-cases/
  Result: Verified
```

Figure 12

Now, after moving to another trivially different testcase, we already find ourselves with another issue. This means that we've properly added support for the previous opcodes, and the next steps would be understanding and adding the semantic knowledge for instructions such as CheckString. This knowledge could be leveraged to add support for other String Operations that are currently unsupported.



```
turbo@turbotv ~/t/test-cases (dev)> ../turbo-tv --verify ./
Invalid Instruction: #59:CheckString[FeedbackSource(INVALID)](#2:Parameter, #45:Call)(#23:Checkpoint)()
Cannot parse operands

Fatal error: exception Lib.Instr.Invalid_instruction("#59:CheckString[FeedbackSource(INVALID)](#2:Parameter,
 #45:Call)(#23:Checkpoint)()", "Cannot parse operands")
Raised at Lib__Instr.err in file "lib/parser/instr.ml", line 20, characters 2-45
Called from Lib__Instr.create_from.parse_operands.parse_operand in file "lib/parser/instr.ml", line 105, cha
racters 16-54
```

Figure 13

Discussion

Future work continues as what Turbo-TV proposed in their discussion. For our work, adding support for more opcodes is the next step. For reference, CheckString was the next invalid instruction that needed its semantic information added for operand parsing. Ultimately, we hope to support JSCall for the base case of function inlining. The function inlining optimization pass is an initial pass to target for its simplicity. This would serve as a building block to validating interprocedural function calls, which would greatly increase the corpus of programs that TurboTV can validate.

With the addition of more opcodes and more supported passes, more coverage is gained. More coverage into the space of possible JS programs generatable with a tool such as Fuzzilli [7]. This added coverage and increased semantic knowledge, which ends up aiding equivalency checks, allowing more of these programs to be tested. This lends itself to higher value fuzzing and more robust translation validation.

References

1. S. Kwon et al., “Translation Validation for JIT Compiler in the V8 JavaScript Engine,” in Proc. IEEE/ACM 46th Int. Conf. Softw. Eng., 2024.
2. N. Lopes et al., “Alive2: bounded translation validation for LLVM,” in Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implement., 2021, pp. 65–79.
3. F. Brown et al., “Towards a verified range analysis for JavaScript JITs,” in Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implement., 2020, pp. 135–150.
4. D. Demange, Y. Retana, and D. Pichardie, “Semantic reasoning about the sea of nodes,” in CC 2018 – 27th Int. Conf. Compiler Constr., 2018, pp. 163–173.
5. StatCounter Global Stats, “Browser Market Share.” Available: https://gs.statcounter.com/browser-market-share
6. Power of Community, “s-92443.pdf,” 2025. Available: https://powerofcommunity.net/2025/slide/s-92443.pdf
7. Power of Community, “Samuel Talk Page,” 2025. Available: https://powerofcommunity.net/2025/talk/samuel.html
8. Google Project Zero, “Fuzzilli.” Available: https://github.com/googleprojectzero/fuzzilli
9. Mozilla Developer Network, “Sloppy mode.” Available: https://developer.mozilla.org/en-US/docs/Glossary/Sloppy_mode