

Validating the JS Rosetta Stone

Group 5
Chase Killorin and Deepak Kurian

Introduction

- Chrome, Edge, Opera use V8(Javascript Compiler)



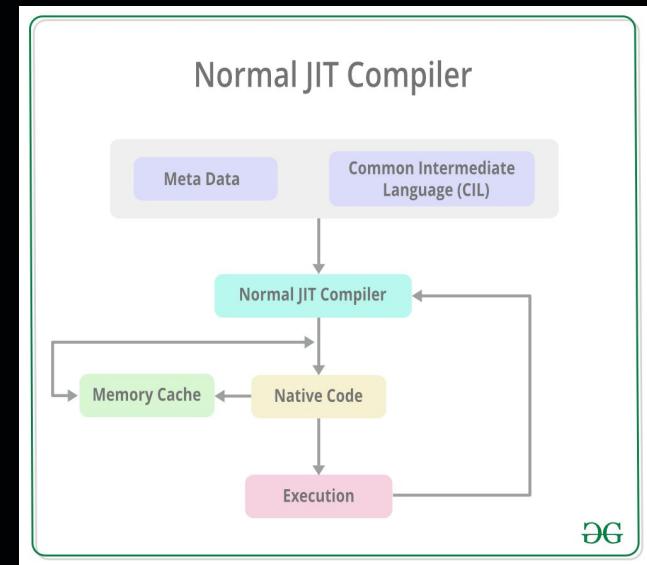
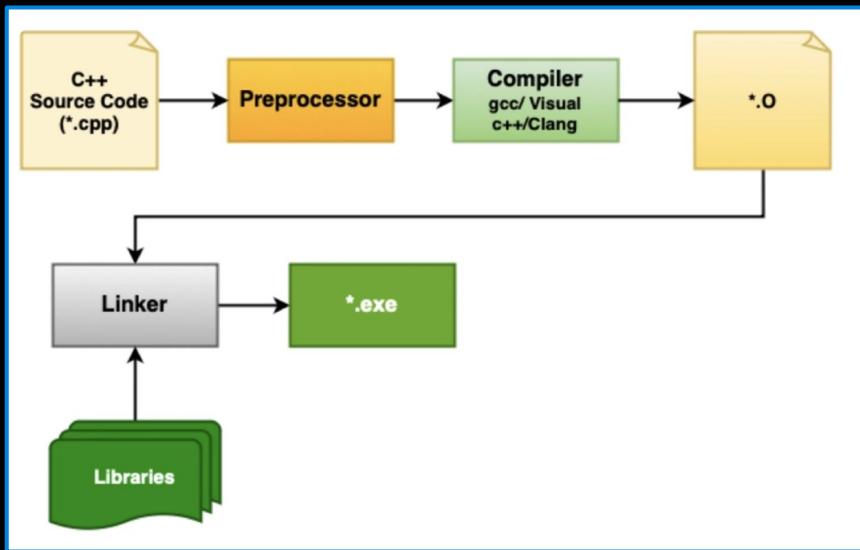
The screenshot shows a browser window with the URL v8.dev/blog/sandbox. The page content discusses memory safety issues in V8, stating: "Memory safety remains a relevant problem: all Chrome exploits caught in the wild in the last three years (2021 – 2023) started out with a memory corruption vulnerability in a Chrome renderer process that was exploited for remote code execution (RCE). Of these, 60% were vulnerabilities in V8. However, there is a catch: V8 vulnerabilities are rarely "classic" memory corruption bugs (use-after-frees, out-of-bounds".

Introduction

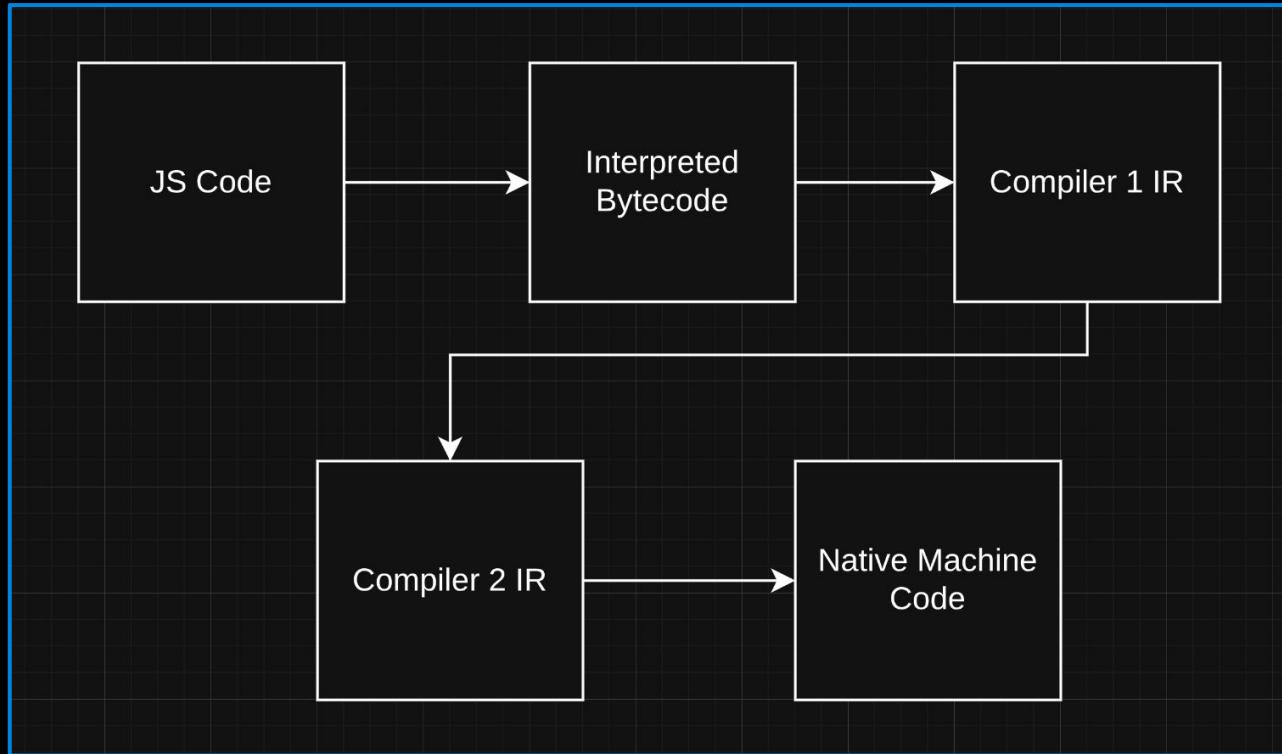
- Research Problem
 - Improve V8 JIT security by improving existing tools (Turbo-TV)
- Research Objectives
 - Supporting Additional Opcodes in Turbo-TV
 - Adding support for the function inlining optimization

Background

- AOT (Ahead of Time) vs JIT (Just in Time) compilation



Background



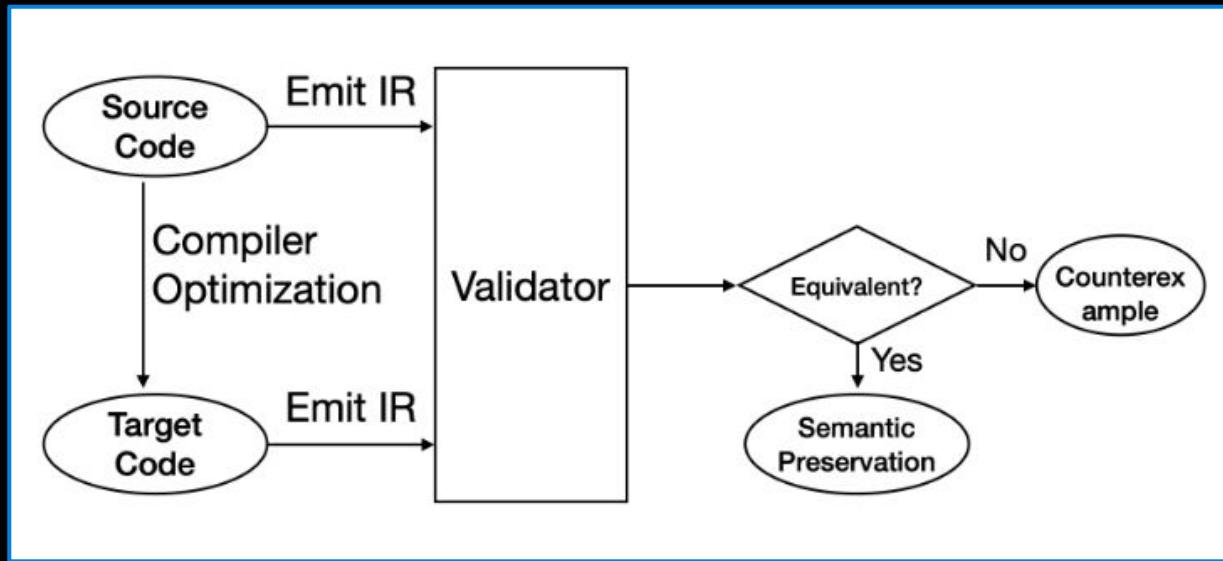
Background - Baseline

Turbo-TV - Translation Validation for JIT Compiler in the V8 JavaScript Engine

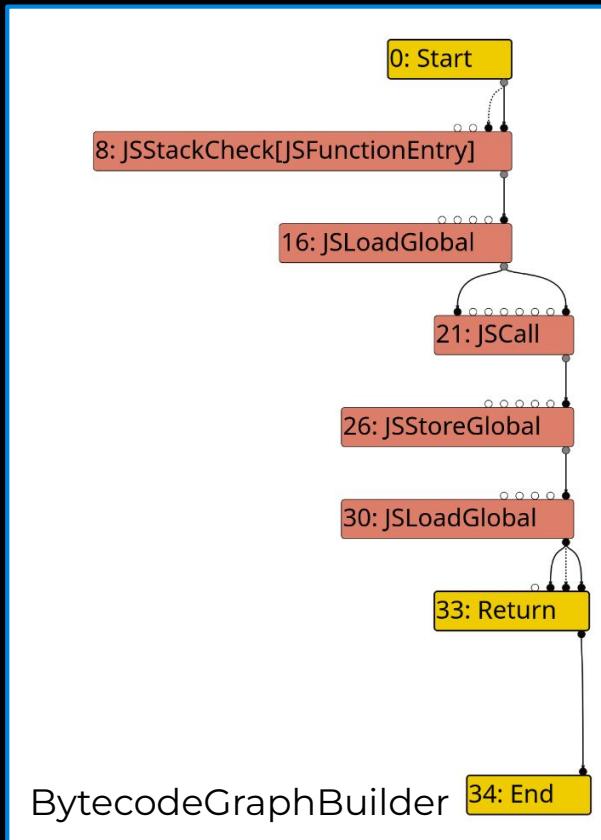
Kwon, S., Kwon, J., Kang, W., Lee, J., & Heo, K. (2024). Translation Validation for JIT Compiler in the V8 JavaScript Engine. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. Association for Computing Machinery.

Background

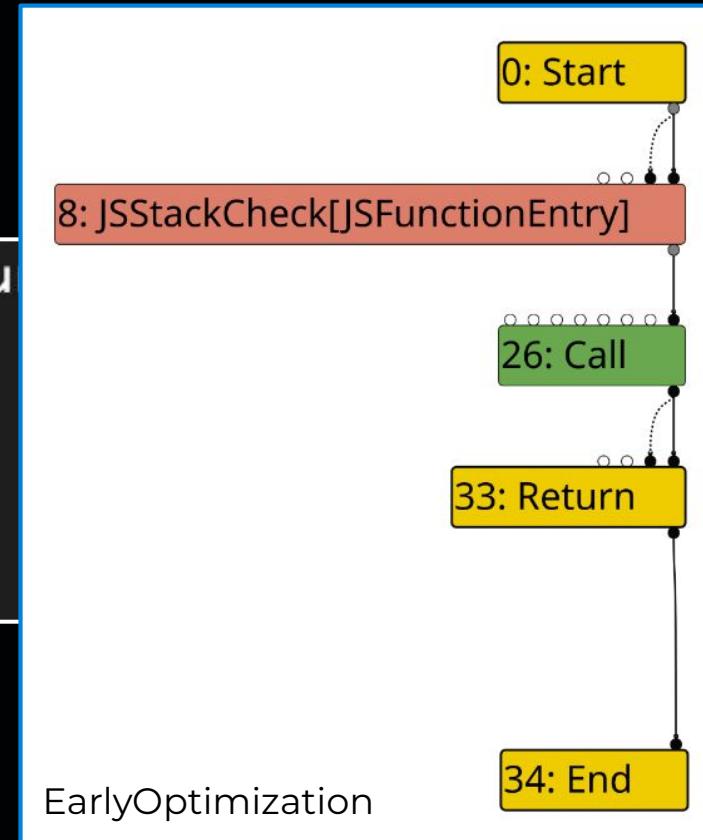
- Translation Validation



Background - Semantic equivalency

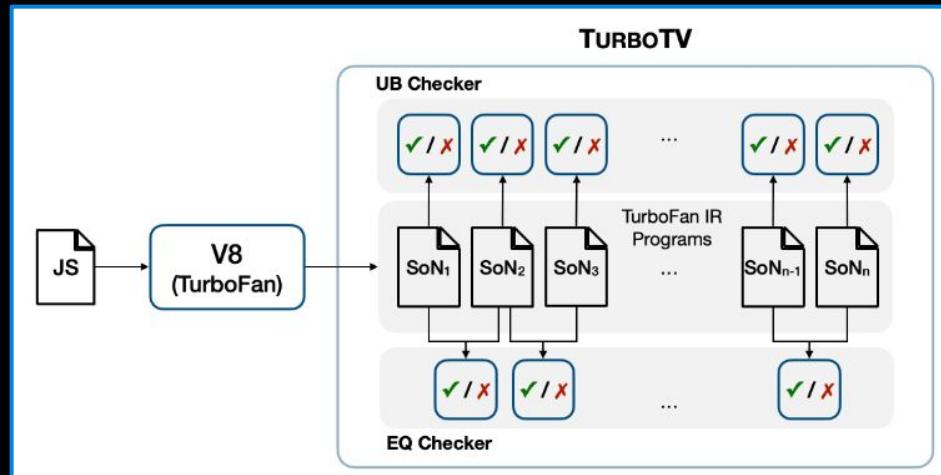


```
(x) { return  
(x) {  
= x+b(x);  
turn y;
```



Background - Turbo-TV Basics

- Undefined Behaviour Checker
- Equivalency checker (Our focus)



Related Work

- Towards a Verified Range Analysis for JavaScript JITs (VeRA)
 - Partial Verification
 - Implemented inside the compiler
- Alive2: Bounded Translation Validation for LLVM
 - Applied to AOT compilation
 - Creates a formalization of intended semantics

Contribution - Turbo TV

- Identifies JIT semantic bugs AOT
- Applies Translation Validation against V8 JIT
- TurboTV constructs a novel adaptation of its prior works
- Difficulty: No formal semantics for SoN IR

Contribution - Turbo TV

- Integrates equivalency checks into Fuzzilli
- Checks don't support everything Fuzzilli can generate

Fuzzilli

A (coverage-)guided fuzzer for dynamic language interpreters based on a custom intermediate language ("FuzzIL") which can be mutated and translated to JavaScript.

Contribution - Turbo TV

Internal State Representation

<i>State</i>	$= Label \times RegFile \times Memory \times Deopt \times UB$
<i>RegFile</i>	$= Register \rightarrow Value$
<i>JSValue</i>	$= TaggedPointer \uplus TaggedSigned$
<i>Value</i>	$= JSValue \uplus Bool \uplus Int8 \uplus Int16 \uplus Int32 \uplus \dots$
<i>TaggedPointer</i>	$= BlockID \times Int32$
<i>TaggedSigned</i>	$= Int31$
<i>Memory</i>	$= BlockID \rightarrow Block$
<i>Block</i>	$= Byte^*$

Methodology

- Lowering Javascript to IR

```
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

Methodology

- Assessing unsupported opcodes

```
~/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:Checkpoint)(#8:JSStackCheck)
Unknown opcode: SpeculativeSmallIntegerAdd

Fatal error: exception Lib.Instr.Invalid_instruction("#50:SpeculativeSmallIntegerAdd[SignedSmall](#2:Parameter, #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
Called from Lib__Instr.create from main.main in file "lib/main/main.ml" line 75 characters 1-22
```

Methodology

- Unimplemented opcodes

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

Methodology

- SpeculativeSmallIntegerAdd

```
(* 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
```

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

Methodology

- JSStoreGlobal

```
| JSStoreGlobal ->
(* B2V1V2E1C1: bracket operands (sloppy mode, global)
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
```

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

Methodology

- Changes

```
'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
```

Results

- Support for:
 - JSStoreGlobal
 - SpeculativeSmallIntegerAdd

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

Results - To be implemented

- Continue to add support for more instructions.
- Ultimately support JSCall for base case of function inlining.

Future Directions

- More opcode support ~> More coverage

```
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
```

Conclusion

- Increase coverage ~> More bugs
- Improved semantic knowledge on certain key Opcodes
 - (JSStoreGlobal)
 - Aids equivalency checks