Translation Validation for JIT Compiler in the V8 JavaScript Engine

Seungwan Kwon
seungwan.kwon@kaist.ac.kr
KAIST
Daejeon, Korea

Jaeseong Kwon
jaeseong.kwon@kaist.ac.kr
KAIST
Daejeon, Korea

Wooseok Kang
kangwoosukeq@kaist.ac.kr
KAIST
Daejeon, Korea

Juneyoung Lee*
lebjuney@amazon.com
Amazon
Austin, Texas, USA

Kihong Heo
kihong.heo@kaist.ac.kr
KAIST
Daejeon, Korea

ABSTRACT

We present TurboTV, a translation validator for the JavaScript (JS) just-in-time (JIT) compiler of V8. While JS engines have become a crucial part of various software systems, their emerging adaption of JIT compilation makes it increasingly challenging to ensure their correctness. We tackle this problem with an SMT-based translation validation (TV) that checks whether a specific compilation is semantically correct. We formally define the semantics of IR of TurboFan (JIT compiler of V8) as SMT encoding. For efficient validation, we design a staged strategy for JS JIT compilers. This allows us to decompose the whole correctness checking into simpler ones. Furthermore, we utilize fuzzing to achieve practical TV. We generate a large number of JS functions using a fuzzer to trigger various optimization passes of TurboFan and validate their compilation using TurboTV. Lastly, we demonstrate that TurboTV can also be used for cross-language TV. We show that TurboTV can validate the translation chain from LLVM IR to TurboFan IR, collaborating with an off-the-shelf TV tool for LLVM. We evaluated TurboTV on various sets of JS and LLVM programs. TurboTV effectively validated a large number of compilations of TurboFan with a low false positive rate and discovered a new miscompilation in LLVM.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; *Just-in-time compilers*; *Semantics*; • **Theory of computation** → *Logic and verification.*

KEYWORDS

Translation Validation, Javascript Engine, JIT Compiler, IR, Semantics, Fuzzing

ACM Reference Format:
Seungwan Kwon, Jaeseong Kwon, Wooseok Kang, Juneyoung Lee, and Kihong Heo. 2024. Translation Validation for JIT Compiler in the V8 JavaScript Engine. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639189

*He contributed to this work before joining Amazon.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639189

1 INTRODUCTION

The correctness of JavaScript (JS) engines (e.g., V8 in Chromium [19]) is one of the most critical issues for the reliability of a wide range of software platforms [25, 27–29, 35]. Recently, the emerging adaption of *Just-in-Time* (JIT) compilers in modern JS engines has made the problem even more challenging. Recently reported bugs demonstrate that the complex nature of JIT compilation often leads to critical miscompilations that can be exploited as a wide range of security vulnerabilities [7–13].

Existing approaches to checking the correctness of the JIT compilers fall into two extremes. One dominant direction is to develop fuzzers that randomly generate JS code and test the engines. They check whether the engines produce crashes [4, 20, 30, 32, 36] or cross-check the outputs of a JS program with and without JIT compilation [4, 36]. While this approach has been widely used in practice, it is not applicable to find latent bugs not observable during the executions of compiled programs. (e.g., crashes or return values). The other direction is to develop a verified JIT compiler from scratch [2, 5]. While this approach can guarantee the correctness of (a part of) the compiler, it incurs substantial effort to rewrite the whole compiler which involves complicated optimizations.

In this paper, we present an *SMT-based translation validation* as a "sweet spot" between the two extremes. Translation validation (TV) checks whether a specific compilation from the source program to the target program is semantically correct [31]. Since we are symbolically checking the semantic preservation using SMT solvers, our technique can discover latent miscompilations during intermediate optimization steps and consider all possible input values of compiled functions. Also, the checking solely relies on the semantics of the source and the target programs and does not require the implementation details of the compiler. This enables us to easily check the correctness even though the compiler is implemented in a complex language like C++ and is updated frequently.

For efficient TV for JS, we propose a novel design of a *staged* strategy. Conventional TV for languages with undefined behaviors (UB) checks a *refinement relation* between a source and a target program [1, 23, 24]. In this work, we carefully rely on the absence of UB in JS based on the ECMAScript specification [18]. This means

that the intermediate programs generated during the JIT compilation do not have UB either if they are correctly compiled. Therefore, we can decompose the whole validation step into two stages: checking the UB of each source and target program and checking the semantic equivalence between the two programs if no UB is found. This strategy enables us to derive simpler SMT queries than the refinement query by the conventional approach.

Furthermore, we leverage fuzzing to achieve practical TV. Since JIT compilation happens at runtime, the overhead of TV degrades the performance of the applications. To address this challenge, we use a fuzzer to generate a large corpus of JS functions that trigger various compiler optimization passes and check the correctness of the JIT compilation for the corpus using TV. This combination enables us to discover latent bugs that are not observable as outcomes. We demonstrate that the cost of TV is amortized to a small fraction of the total running time as the fuzzer runs long enough. We also utilize the fuzzer to test our TV tool itself. We generate a pair of JS functions that return different values given the same input. Then, we check whether our tool can capture the semantic difference.

Finally, we extend our approach to cross-language TV. Combining our tool with an existing TV tool for LLVM IR, Alive2 [24], we check the correctness of the translation from LLVM IR to TurboFan IR. We first translate an LLVM IR program to a WebAssembly (Wasm) bytecode using the LLVM compiler. Then, we translate the Wasm bytecode to TurboFan IR using the TurboFan compiler. Finally, we check the correctness of the compilation from the original LLVM IR program to the translated TurboFan IR program. This naturally enables us to check the correctness of the Wasm backend of LLVM and the Wasm frontend of TurboFan. Since the memory models of LLVM and TurboFan are different, we focus on functions whose parameters and return values are integers or floats. Nevertheless, we found a miscompilation in the Wasm backend of LLVM that cannot be found by the existing TV for LLVM IR.

We instantiated these ideas in a tool TurboTV, a TV for TurboFan (JIT compiler of V8). We evaluated the effectiveness of TurboTV on a large number of JS and LLVM benchmarks, including reported bugs, regression tests, and generated corpus. The results demonstrate that TurboTV is robust enough to discover all the bugs from the benchmarks with a low false positive ratio.

In summary, this paper makes the following contributions:

- We present TurboTV, the first SMT-based TV for TurboFan. We formally define the semantics of TurboFan IR as SMT encoding.
- We present a two-stage TV strategy: UB checking and semantic equivalence checking. This decomposition enables us to derive simpler SMT queries than conventional refinement queries.
- We extend TurboTV to cross-language TV from LLVM to TurboFan using Wasm as an intermediate language. We found a miscompilation in the Wasm backend of LLVM that is not found by the existing TV for LLVM IR.
- We evaluate and demonstrate the effectiveness of TurboTV with a large set of JS and LLVM programs. Our tool and data are available at https://doi.org/10.5281/zenodo.10453785 and https://github.com/prosyslab/turbo-tv-artifact.

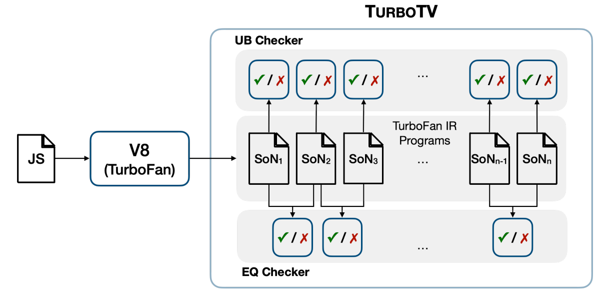


Figure 1: Overview of TurboTV. SoN_i is a graph representation of IR of TurboFan (Sec. 4.1) after i'th reduction. EQ Checker and UB Checker are explained in Sec. 2.3.2.

2 OVERVIEW

2.1 The Sea-of-Nodes IR

In TurboFan, a function is represented in a graph-based IR called Sea-of-Nodes (SoN) [16]. SoN is a directed graph where each node represents an operation or a constant, and each edge represents a data or control dependency. Notice that SoN does not explicitly specify execution orders. That is, two nodes can be executed in arbitrary order if there are no edges in between. After all optimizations are applied, TurboFan generates a conventional control-flow graph (CFG) from SoN by explicitly specifying the execution order.

2.2 Translation Validation (TV)

We briefly describe the compiler correctness and TV. To validate a compiler transformation, one needs to check whether the semantics of the source program is preserved in the target program. Semantic preservation is defined as a *refinement* relation between the source and target programs' behavior [31]. Given a pair of behavior (B_1, B_2), B_2 refines B_1 if (1) B_1 and B_2 are well-defined and equal[1], or (2) B_1 is not well-defined; in other words, it is undefined behavior (UB). UB is the behavior of a program that does not satisfy the type checker or language standard.

A validator takes a pair of programs – the source program and the target program – and checks whether the refinement relation holds between the behavior of the source and target program for every input. For validation of intraprocedural compiler transformation, the functions in the source and target programs are aligned by their names, and each function pair with the same function name is validated. A validator symbolically encodes the final states of the two functions for a function input. In this paper, we will call the symbolic final state the semantic of the function.

2.3 Our Approach: TurboTV

2.3.1 Goal of TurboTV. The goal of TurboTV is to validate intraprocedural optimizations of loop-free functions. Given a JS function, TurboTV validates all the optimization steps (called *reductions*) during the JIT compilation. It works with TurboFan, which is specially instrumented to emit the IRs of the source and target

[1]Note that this definition of refinement does not consider nondeterminism for brevity. If nondeterminism is considered, this equality must be expanded to the subset relation of two behavior sets. Since nondeterminism is rare in JS, we only consider deterministic behavior (see Sec. 5.3).

functions for each optimization step. Then, TurboTV symbolically executes the functions and emits verification conditions that encode compiler correctness. Finally, the SMT solver checks the verification condition. The role of the solver is to find an input to the functions that breaks the condition for compiler correctness. The overall architecture of TurboTV is shown in Fig. 1.

Since compilation speed is important for JIT compilers, running TurboTV for every compiling program might not be the best option. Instead, TurboTV can be used as a way of testing TurboFan with wider test coverage compared to traditional random testing. In Sec. 7 and 8, we show that TurboTV can be effectively combined with fuzzing at a small cost. Specifically, we demonstrate that the cost of TV is amortized to a small fraction of the running time when the fuzzer's running time becomes long enough.

2.3.2 *EQ Checker and UB Checker.*

According to the ECMAScript specification [18], JS programs do not have UBs as in C/C++[2]. This fact enables us to decompose the refinement checking into two stages: EQ (equality) check and UB check. We name this a two-stage TV strategy.

The underlying principle is as follows. Let's assume that $f_{\mathrm{JS}}(x)$ and $f_{\mathrm{IR}}(x)$ are a JS function and its IR, respectively. We assume that the translation only looks into $f_{\mathrm{JS}}(x)$ (i.e., intraprocedural). Since (1) $f_{\mathrm{JS}}(x)$ does not raise UB for any input x, and (2) the compiler must not introduce UB according to the definition of refinement, f_{IR} also does not raise UB for any IR value x that represents some valid value in JS.

Now, let us assume that $f_{\mathrm{IR}}(x)$ is optimized to $f'_{\mathrm{IR}}(x)$ via an intraprocedural optimization. If the optimization was correct, f'_{IR} again must not have UB for any valid input x. After proving that f'_{IR} has well-defined behavior for any x, showing the correctness of optimization is finally reduced to simply showing the equivalence of the behavior of f_{IR} and f'_{IR} for any valid x. Note that the two checks — the existence of UB and behavior equivalence — can be naturally done via two independent checkers. Therefore, we split the validation into invocations of two different checkers: UB Checker and EQ Checker.

The UB Checker inspects whether a given IR function does not raise UB for any valid input. In our formal semantics of TurboFan IR, erroneous behavior such as out-of-bounds access is regarded as UB (see Sec. 3). The UB Checker of TurboTV detects compiler bugs introducing such behavior. The EQ Checker takes two TurboFan functions that are before and after a reduction (optimization step) and proves that they are semantically equivalent.

The separation of UB and EQ Checkers has two benefits. First, the split SMT queries are shorter than the original refinement query, providing more opportunities for the SMT solver to answer within a given resource. One refinement query is split into two queries for UB Checkers and one query for EQ Checker. The two queries for UB Checkers are the conditions of UB of source and target functions. If consecutive compiler transformations are validated, the results of the UB Checker for the target function can be reused for validation of the next transformation.

Second, it effectively detects miscompilation bugs introducing UB. Consider a TurboFan function f_{IR} that raises UB for some valid JS value x_0 as an input. The fact that $f_{\mathrm{IR}}(x_0)$ raises UB implies that there is a miscompilation during a series of compiler transformations from the source JS program f_{JS} to f_{IR} because JS does not have UB. In theory, validating every transformation with the conventional refinement relation will detect where the UB was introduced. However, the bug can be missed if the refinement checking fails due to some practice limitations, such as the solver's timeout. Instead, in two-stage TV, UB Checker can detect the bug by directly inspecting $f_{\mathrm{IR}}(x)$ only. We show that TurboTV does not miss bugs in Sec. 8.

[2]The specification does not specify UB. Notice that *implementation-defined* behavior in ECMAScript is well-constrained by the specification and different from UB.

2.4 Validation Scope of TurboTV

We consider the behavior *after deoptimization* out of the scope of this paper. JIT compilers optimize the code based on specific assumptions about the input. Once the assumptions are invalidated during the execution, deoptimization is triggered, and the function is executed by the interpreter. Since our goal is to check the correctness of the JIT compiler, we prove the semantic equivalence between the source and target functions for all inputs that do not invoke deoptimization.

Since our scope is validating intraprocedural optimizations, TurboTV may falsely report that miscompilation happened after *inter*procedural optimizations. Furthermore, a single existence of interprocedural optimization may cause the UB Checker to raise false alarms for all later optimization because it may introduce assumptions that rely on global invariants. However, we experimentally show that TurboTV has very low false alarms in practice, as other SMT-based validators do.

TurboTV only supports loop-free functions. Modeling the semantics of a function, including possibly unbounded loops, and validating their transformations is known to be a hard problem. We leave this extension as future work.

3 MOTIVATING EXAMPLES

We illustrate our approach with two real bugs of V8. We will first explain the validation process of the EQ Checker in Sec. 3.1, then describe the details of the UB Checker in the rest of the section.

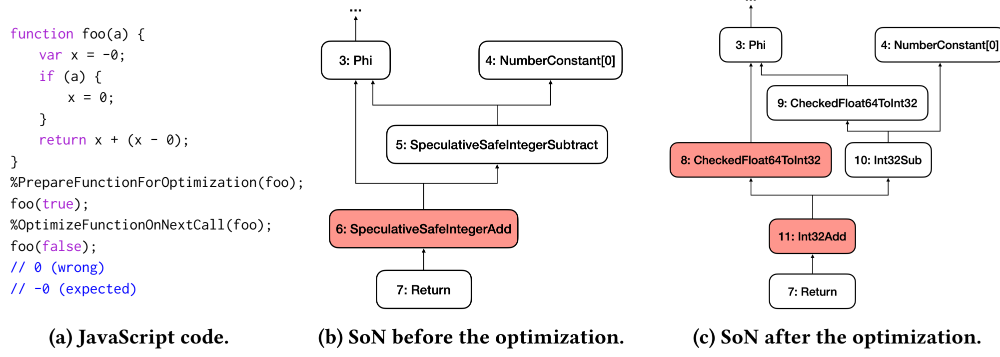
3.1 A Miscompilation Bug: Issue 1199345

3.1.1 *Bug Description.*

Fig. 2(a) shows a JS code that triggered a miscompilation due to the incorrect handling of signed zero (-0) in JS [10]. Function `foo` negates the value of `x` if argument `a` is true and returns `x + (x - 0)`. If `a` is true, the function returns $0 + (0 - 0) = 0$. Otherwise, the function returns the result of $(-0) + ((-0) - 0) = -0$ according to the ECMAScript specification [18].

TurboFan optimizes `foo` using the calling context described between lines 8 and 10. The macros at lines 8 and 10 force the compiler to optimize the function for the next call at line 11. After the first call to `foo` (line 9), TurboFan speculatively optimizes the function based on the input value (`true`).

We will explain how TurboFan miscompiled `foo` to return `0` even if `a` was `false` by presenting its TurboFan IRs before and after a problematic compiler optimization. Fig. 2(b) depicts an IR of the function before the optimization is run. `SpeculativeSafeIntegerAdd` and `SpeculativeSafeIntegerSubtract` compute the addition and subtraction of two floating-point operands if both are *safe* integers, meaning that it is in $[-2^{53}+1, 2^{53}-1]$. Otherwise,



(a) JavaScript code. **(b) SoN before the optimization.** **(c) SoN after the optimization.**

Figure 2: Issue 1199345 of Chromium [10].

they trigger deoptimization and TurboFan fallbacks to running V8's JS interpreter. Since 0 and -0 are considered safe integers, the function computes a correct output without deoptimization, regardless of the input.

Fig. 2(c) depicts the IR after the function is optimized to use `Int32Add`. This function is faster than the original code since `Int32Add` uses the simple 32-bit integer addition, whereas `SpeculativeSafeIntegerAdd` internally uses floating-point addition. However, it is incorrect because `CheckedFloat64ToInt32` does not trigger a deoptimization when its operand is -0. It deoptimizes when the operand is not representable in a 32-bit integer without *loss*, but TurboFan does not consider the conversion of -0 to 0 as a lossy. Therefore, -0 is cased to 0 if `a` is false, and eventually, the function returns 0. A correct compilation is to use `CheckedFloat64ToInt32`$_{-0}$ which has a special mode `CheckForMinusZero` to trigger a deoptimization when the operand is -0.

3.1.2 Validation via SMT Solving. TurboTV validates the miscompilation of TurboFan via SMT solving. The main idea is to symbolically encode the semantics of each IR function and check whether the source's semantics is preserved after the optimization. Let r_i denote the result value of the instruction at node i and d_i indicate whether a deoptimization has been triggered until node i before the optimization. Similarly, we denote the result value and deoptimization flag at node i after the optimization by r'_i and d'_i, respectively. For example, the semantics of the red nodes in Fig. 2 are represented as follows:

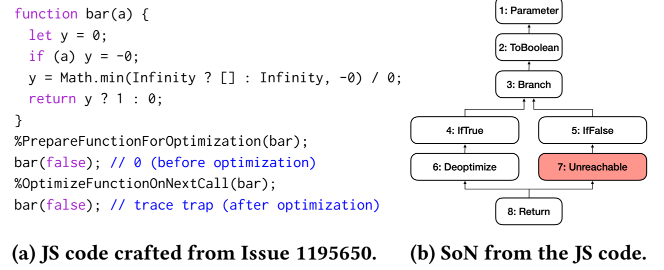
Node 6 in Fig. 2(b) The result r_6 is computed as the addition of the results r_3 and r_5 from the previous instructions: $r_6 = r_3 + r_5$. A deoptimization is triggered if either r_3, r_5 or r_6 is not a safe integer: $d_6 = d_3 \lor d_5 \lor \neg(\mathsf{IsSafeInt}(r_3) \land \mathsf{IsSafeInt}(r_5) \land \mathsf{IsSafeInt}(r_6))$.

Node 7 in Fig. 2(b) The return instruction just outputs the incoming value and the deoptimization flag: $r_7 = r_6$ and $d_7 = d_6$.

Node 8 in Fig. 2(c) This operator casts the operand into an `Int32` value: $r'_8 = \mathsf{ToInt32}(r'_3)$. A deoptimization is triggered when the type cast is lossy: $d'_8 = d'_3 \lor (\mathsf{ToFloat}(r'_8) \neq r'_3)$. Note that the conversion from -0 to 0 is lossless.

Node 11 in Fig. 2(c) The result is addition of the operands r'_8 and r'_{10} and the deoptimization flag is set if any of the previous instructions triggered deoptimizations: $r'_{11} = r'_8 + r'_{10}$ and $d'_{11} = d'_8 \lor d'_{10}$.

Node 7 in Fig. 2(c) The return instruction just outputs the incoming value and the deoptimization flag: $r'_7 = r'_6$ and $d'_7 = d'_6$



(a) JS code crafted from Issue 1195650. **(b) SoN from the JS code.**

Figure 3: Issue 1195650 of Chromium [8].

Finally, EQ Checker checks whether the return values r_7, r'_7 are equal for any input 'a'. This is done by finding 'a' that satisfies the negated condition using an SMT solver: $(r_7 \neq r'_7) \land \neg d_7 \land \neg d'_7$. When a = false, r_7 and r'_7 have -0 and 0 without deoptimizations, respectively. Thus, EQ Checker reports this mistransformation.

3.2 A Miscompilation Bug: Issue 1195650

The previous example demonstrates how the symbolic formula of the correctness of the compilation is written. Now, we will move to a slightly more complicated bug that involves UB.

3.2.1 Bug Description. Fig. 3(a) shows a JS code crafted from another miscompilation issue [8]. The value of `y` at line 4 is *NaN* because it divides -0 with 0. Since *NaN* is interpreted as false in JS, the return value is always zero.

However, TurboFan incorrectly optimizes the code and generates an IR with UB. Fig. 3(b) shows the simplified version of the IR. In TurboFan IR, deoptimization is triggered if an argument of `Math.min` is not a number type value. In this example, the first argument of `Math.min` is the empty array (`[]`) which is not a number type value. Thus, this function always triggers a deoptimization regardless of which branch is taken at line 3. TurboFan should have inserted the instruction `Deoptimize`, which explicitly triggers a deoptimization to both of the branches. After that, node Unreachable should have been inserted after `Deoptimize` to indicate that the rest of the code is dead. However, TurboFan incorrectly inserts `Unreachable` as shown in the figure. Therefore, the compiled code does not trigger a deoptimization when the input is false but executes invalid code that leads to SIGTRAP.

3.2.2 Validation via SMT Solving. TurboTV considers the reachability to `Unreachable` as UB. We check whether an input exists

that makes the execution reachable to the node. Similar to the deoptimization flag, we denote the UB flag at node i by u_i that is set when a UB is triggered. Here is the SMT encoding to detect the UB of bar in Fig. 3(b):

Node 1 Suppose p is the parameter value that can be an arbitrary value. Initially, the UB flag and deoptimization flags are not set: $r_1 = p$ and $u_1 = d_1 = \text{false}$.

Node 2 The result r_2 is a boolean value obtained by converting r_1 according to the ECMA Specification. This node does not trigger UB and deoptimization; the flags are copied from node 1: $r_2 = \text{ToBool}(r_1)$, $u_2 = u_1$, and $d_2 = d_1$.

Node 3, 4, and 5 The nodes do not evaluate any value but propagate the flags: $u_5 = u_4 = u_3 = u_2$ and $d_5 = d_4 = d_3 = d_2$.

Node 6 The function triggers a deoptimization if this node is reachable. That is, the branch condition is true , and no UB has been triggered before the node: $d_6 = \text{IsTrue}(r_2) \land \neg u_4$. The UB flag is propagated from the previous node if the branch condition holds: $u_6 = \text{IsTrue}(r_2) \land u_4$.

Node 7 Similarly, we set the UB flag only when this node is reachable. In this case, we consider the node to be reachable if the branch condition is false and no deoptimization has been triggered before the node: $d_7 = \text{IsFalse}(r_2) \land d_5$ and $u_7 = \text{IsFalse}(r_2) \land \neg d_5$.

Node 8 The flags are set if a deoptimization or a UB is triggered along with the true or false branch: $d_8 = d_6 \lor d_7$ and $u_8 = u_6 \lor u_7$.

Finally, the UB Checker checks whether condition $\neg d_8 \land u_8$ holds. The condition is satisfiable if there is an input that triggers a UB during the execution before triggering any deoptimizations.

In our operational semantics, there are three cases of erroneous operations that have UB: (1) execution of the Unreachable node, (2) out-of-bound memory access, and (3) the execution of a node that is annotated with incorrect range information. The last case happens when V8's range analysis is buggy, and it is also the case where our SMT-based approach has a benefit compared to the fuzzer approach. A wrongful range annotation is not externally observable unless a later compiler transformation utilizes the range information and transforms the function into a crashing one. This condition makes it hard for fuzzers to detect bugs in V8's range analysis. Our SMT-based approach can detect such bugs well because it does not rely on the optimization pipeline.

Given the absence of a formal specification for TurboFan IR, our definition of UB is grounded in a set of criteria. Firstly, we conducted an analysis of known security bugs in V8, identifying their root causes. Secondly, we cross-referenced these behaviors with the classification of UB in LLVM IR. Finally, employing a UB checker based on our definition, we conducted experiments to confirm that our definition accurately captures the erroneous behavior of TurboFan.

4 FORMAL SEMANTICS OF TURBOFAN IR

In this section, we introduce the IR of TurboFan (Sec. 4.1 and 4.2). Also, we introduce the formal semantics of TurboFan IR defined by us (Sec. 4.3). Formal semantics is used to symbolically encode the final states of the given source and target functions, which are also described in the previous section's examples.

4.1 The Sea-of-Nodes IR

A SoN function is a directed labeled graph $G_S = \langle Node, \rightarrow_S \rangle$. Each node has a unique label and is associated with an instruction. We assume an auxiliary function $\text{inst}(l)$ that provides the instruction of a given node label l. A directed edge between two nodes means that a dependency exists between the instructions. An edge is either a data edge, control edge, or effect edge. Data edges represent data dependencies of registers and constants. Control and effect edges specify control dependencies introduced by control instructions (e.g., branch) and side effects (e.g., load/store), respectively.

In TurboFan, there are four categories of instructions that are distinguished by in which compilation stage they appear: JS, Simplified, Machine, and Common. Instructions in the JS category appear immediately after the JS code is translated into the TurboFan IR. As operators in JS do, they can take any type of argument. Then, TurboFan converts some JS instructions into Simplified instructions that are different from JS in two aspects. First, Simplified instructions are aware of the precise memory layout of each object and use primitive loads and stores to manipulate their fields. Second, a typical Simplified instruction is specialized for a specific input type. For example, `SpeculativeNumberAdd` is an addition that is specialized for numbers. If inputs do not have the number type in JS, it triggers deoptimization. Finally, the category at the lowest level is Machine, whose instructions can be easily translated into the assembly language. There is the last category called Common, which contains instructions that can be shared across all levels such as nodes for describing conditional branches.

4.2 Scheduling and Validity of Sea-of-Nodes

After all optimizations, the SoN graph is *scheduled* so that every instruction has execution order. We simply call a scheduled SoN graph as a *control-flow graph* (CFG). Given a SoN IR $G_S = \langle Node, \rightarrow_S \rangle$, a CFG $G = \langle Node, \rightarrow_C \rangle$ consists of the same set of nodes ($Node$) and the control-flow edges ($\rightarrow_C$) between the nodes which may differ from $\rightarrow_S$.

If G_S does not specify total ordering between its nodes, there may exist multiple possible schedules for G_S. In such cases, V8 simply assumes that all the scheduled programs must be semantically equivalent and derives a *well-ordered* CFG that preserves all dependencies in G_S. Given a SoN $G_S = \langle Node, \rightarrow_S \rangle$, a CFG $G = \langle Node, \rightarrow_C \rangle$ derived by a scheduling is well-ordered if

$$\forall c_1, c_2 \in Node.\ c_2 \rightarrow_S c_1 \implies c_1 \rightarrow_C^+ c_2.$$

Intuitively, if c_2 depends on c_1 according to G_S, there must exist a path from c_1 to c_2 in G.

The validity of SoN, representing the V8's assumption, is defined using the above definition. A SoN graph G_S is valid if all the well-ordered CFGs scheduled from G_S are semantically equivalent. V8 assumes that creation of an invalid G_S is miscompilation.

For TV, TurboTV first checks the validity of input SoNs and then chooses one of the well-ordered CFGs for subsequent validation. The details of the validity checking will be described in Sec. 5.2.

4.3 Formal Semantics

We define the formal semantics of TurboFan IR in an operational style. Strictly speaking, we define the formal semantics of program execution of a control-flow graph G rather than SoN G_S.

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

Figure 4: Semantic domains. *JSValue* is a set of values in JS, whereas *Value* is a set of values used by TurboFan internally. *TaggedPointer* and *TaggedSigned* has prefix 'Tagged' because they are distinguished by a tag bit in the V8 internal.

Fig. 4 shows the definition of semantic domains. A program state $S = \langle l, R, M, D, U \rangle$ is a tuple of a node label, a register file, a memory, a deoptimization flag, and a UB flag. *RegFile* is a mapping from registers to values. We consider common types of values in TurboFan IR. Memory is a mapping from block IDs to memory blocks. Each memory block is an array of bytes. The deoptimization and UB flags at a state indicate that the program has triggered deoptimizations and executed UB.

The semantics for each instruction is specified with a transition relation. We denote $(\hookrightarrow) \subseteq S \times S$ as the transition relation between two states. Among TurboFan's various operations, we formalize a subset of Common, Simplified and Machine. Among Common operations, we formalized function prologue and epilogue, constants, branches, function calls, deoptimization, exception throw, and unreachable. For Simplified, we formalized operations on Boolean, BigInt, String, Numbers, and memory operations. For Machine, we formalized arithmetic, bit-wise operations, and memory operations. Fig. 5 shows the semantics for selected instructions.

A parameter value $v_{\mathtt{param}}$ is valid (ValidParam in Fig. 5) if it is a valid representation of some value in JS. Note that every JS value is either a *TaggedSigned* or *TaggedPointer* value in TurboFan. Integer values that can be stored in 31-bit are typed with *TaggedSigned*. All the other values are stored as heap objects, and the referring *TaggedPointer* represents the value [21]. If v is *TaggedPointer*, v may point to many kinds of heap objects. We constrain its referred object to be either (1) floating-point values, (2) basic constants such as `undefined`, (3) string values or (4) big-int values.

5 ENCODING SEMANTICS AND COMPILER CORRECTNESS IN SMT

This section describes our SMT encoding scheme for the semantics of TurboFan IR and the compiler correctness. We only consider programs with a single function definition without loops and function calls to user-defined functions.

5.1 Encoding of Value and Memory

5.1.1 Value. We represent a value (an element of *Value* set) as a 69-bit-vector in SMT. The most significant five bits represent the type of the value. Note that five bits (hence 2^5 values) are necessary because we consider 22 types, 18 types of which are in the latest version of TurboFan and 4 types are deprecated but supported for backward compatibility. The *Value* set consists of the union of the 22 sets, which is omitted in Figure 4 for brevity. The remaining 64 bits encode the actual value according to the type. For example, for `int32` type values, 32 least significant bits of the vector are used. Also, for `float64` type values, we encode the value in IEEE-754 double-precision format. For function parameters, we encode the well-formedness of the inputs described in Sec. 4.3 as an assertion for each parameter. This restricts the SMT solver to find the function inputs that only satisfy the criteria.

The operations in TurboFan are encoded to process inputs and outputs as values in a 69-bit-vector. Taking the `NumberAdd` operator as an example, which adds two floating-point numbers, we encode it to convert inputs into floating-point expressions, perform the addition, and then convert the result back into a 69-bit-vector value. This resultant value serves as the output for subsequent operations.

5.1.2 Memory. We define a memory as a set of memory blocks. Conceptually, a memory block corresponds to an object in Javascript. A memory block contains bytes that describe the contents of the object. We distinguish each memory block by assigning its unique block ID, which is a non-negative integer. We encode memory with two SMT arrays named Bytes and Bsize. Bytes is an SMT array from TaggedPointer which is a 32-bit-vector to a byte which is an 8-bit-vector. Bsize maps a block ID to the size of each block.

TurboFan IR has a pointer that has an address to an object. A pointer value, TaggedPointer, is defined as a 32-bit-vector variable in SMT. According to [21], the maximum size of the memory can be reasonably bounded to 4GiB. We use the high 8 bits of TaggedPointer to describe the block ID and the low 24 bits as the block offset. This implies that our validator may miss a bug if the bug requires using more than 2^8 memory blocks or a single block larger than 2^{24} bytes. This is reasonable because the size of programs fuzzer creates typically has a much smaller number of possibly distinct pointer values than that. We will describe the limitations due to approximations in Sec. 5.3.

As for the input parameters, we encode the well-formedness precondition of loaded values in SMT as assertions. Also, we predefine a few memory blocks as memory blocks containing constants in JS such as `null`, `true` and `false`.

5.2 Encoding Compiler Correctness

This section discusses the encoding of compiler correctness. The validation process of TurboFan consists of two steps. Given a pair of source and target SoN IRs, TurboTV first checks the validity of the IRs as described in Sec. 4.2. Once both of the input SoN are proven to be valid, TurboTV derives two well-ordered CFGs, each of which is scheduled from the source and target SoNs. Finally, TurboTV verifies the refinement relation between the two CFGs using the UB Checker and EQ Checker, We will provide a detailed description of each step in the following subsections.

5.2.1 Validity of SoN. Let us denote the set of CFG as $\mathcal{G}$. We define the execution of a CFG as a function Exec: $\mathcal{G} \times \mathit{State} \longrightarrow \mathit{State}$ that takes a CFG and an initial state as inputs and returns the final state. The set *State* is defined in Sec. 4.3.

Now we formulate the validity of SoN in Sec 4.2. Given a SoN G_S, let $\mathcal{G}_{G_S}$ be the set of well-ordered CFGs scheduled from G_S. We define G_S to be valid if and only if the following condition holds: $\forall G_i, G_j \in \mathcal{G}_{G_S}.\ \forall S \in \mathit{State}.\ \mathsf{Exec}(G_i, S) = \mathsf{Exec}(G_j, S)$.

$$\frac{\text{inst}(l) = \text{“}r = \textbf{Parameter}\text{”} \quad l \to_C l' \quad \text{ValidParam}(v_{\text{param}})}{\langle l,R,M,D,U\rangle \hookrightarrow \langle l',R\{r \mapsto v_{\text{param}}\},M,D,U\rangle}\ \textsc{Param}$$

$$\frac{\begin{array}{c}\text{inst}(l) = \text{“}r = \textbf{SpeculativeSafeIntegerAdd}\ e_1\ e_2\text{”} \\ l \to_C l' \quad v_1 = [[e_1]]_R \quad v_2 = [[e_2]]_R \quad v = v_1 + v_2 \\ d = \neg(\text{IsSafeInt}(v_1) \lor \text{IsSafeInt}(v_2) \lor \text{IsSafeInt}(v))\end{array}}{\langle l,R,M,D,U\rangle \hookrightarrow \langle l',R\{r \mapsto v'\},M,D \lor d,U\rangle}\ \textsc{SpecSafeIntAdd}$$

$$\frac{\begin{array}{c}\text{inst}(l) = \text{“}r = \textbf{CheckedFloat64ToInt32}_{-0}\ e\text{”} \\ l \to_C l' \quad v = [[e]]_R \quad v' = \text{ToInt32}(v) \\ d = \neg(\text{IsInt32}(v) \lor v = -0.0)\end{array}}{\langle l,R,M,D,U\rangle \hookrightarrow \langle l',R\{r \mapsto v'\},M,D \lor d,U\rangle}\ \textsc{CheckedF64ToI32-MinusZero}$$

Figure 5: Semantics of selected instructions. v_{param} is the value of the parameter and the predicate ValidParam states its validity.

Notice that, to show the validity, it is enough to prove that all effect edges are well-established. Recall that a SoN edge is either a data, control, or effect edge. The data and control edges are well-established by the construction of the CFG. Hence, we only need to prove the validity of the effect edges that represent dependencies introduced by side-effects such as memory load and store.

Let us consider two nodes in a SoN, denoted as n_i and n_j, wherein each node contains an operator accessing the same memory address. To have a unique execution result regardless of scheduling, one must depend on the other along the SoN edges ($\to_S$) if (1) both of them perform write operations to the same memory address or (2) one of them performs a write operation, and the other performs a read operation to the same memory address. Then, we ensure that all the scheduled CFGs from the SoN are semantically equivalent. As a result, we can simplify the validity condition as follows:

$$\forall n_i, n_j \in Node_e.\ \text{Reach}(n_i, n_j) \land \text{Overlap}(n_i, n_j) \implies (n_i \to_S {}^+ n_j)$$

where $Node_e$ is the set of all nodes in the G_S whose operators have side-effects. In this condition, $\text{Reach}(n_i, n_j)$ is true if and only if n_j is reachable from the starting node through n_i. Given a SoN $G_S = \langle Node, \to_S\rangle$, $\text{Reach}(n_i, n_j)$ is defined as follows: $\text{Reach}(n_i, n_j) \iff n_0 \to_S {}^* n_i \to_S {}^+ n_j$ where n_0 be the starting node of the G_S. Next, $\text{Overlap}(n_i, n_j)$ is true if and only if the memory regions accessed by executing n_i and n_j can overlap. Let $\text{Access}(n)$ be the set of block IDs accessed at node n. Then $\text{Overlap}(n_i, n_j)$ is defined as follows:

$$\text{Overlap}(n_i, n_j) \iff \exists bid \in BlockID.\ bid \in \text{Access}(n_i) \cap \text{Access}(n_j).$$

In summary, TurboTV encodes the negation of the simplified validation condition as an SMT query. If the query is satisfiable, it means that there exists a pair of nodes that can affect each other but are not ordered in the G_S. In this case, TurboTV reports the given IR as invalid. If the validity is proven, TurboTV selects well-ordered CFGs scheduled from both the source and target and then proves the refinement relation between them.

5.2.2 Refinement. From an input state S, we symbolically encode the final state of source function $f_{src}(S)$ and target function $f_{tgt}(S)$ by iteratively following our operational semantics (Sec. 4.3). To deal with conditional branches, we track the reachability of instruction n from the function entry, say $\text{Reach}(n)$, which is a boolean expression in SMT. Then, the final state S' holds the following constraint: $\bigwedge_i \text{Reach}(\textbf{return}_i) \implies S' = S_{\textbf{return}_i}$ where $\textbf{return}_i$ is i'th **return** node in the function and $S_{\textbf{return}_i}$ is the state at the point. Also, we encode a set of preconditions for the input state $\text{Pre}(S)$ that are described in Sec. 4.3.

Now, we explain the verification conditions of UB Checker and EQ Checker. The UB Checker's verification condition is

$$\forall S\,.\ (\text{Pre}(S) \land \neg f(S).D) \implies \neg f(S).U$$

where $.U$ and $.D$ mean the UB and deoptimization flag of a state. To turn this into a satisfiability problem, we use the negated formula. We call UB Checker for functions f_{src} and f_{tgt}. The verification condition of the EQ Checker – semantic equivalence – is as follows:

$$\forall S\,.\ (\text{Pre}(S) \land \neg f_{src}(S).D \land \neg f_{tgt}(S).D) \implies f_{src}(S) = f_{tgt}(S).$$

This condition is also negated for the SMT solver.

5.3 Approximation in the Encoded IR Semantics

5.3.1 Approximated Arithmetic Operations. We approximate common arithmetic functions that are expensive to encode exactly in SMT. For math operations like $\sin(x)$ and $\cos(x)$, we encode them as an if-then-else expression that returns values for some inputs such as 0 for $\sin(0)$ and returns any value for all other inputs. For the unknown inputs, we use UF (uninterpreted functions) in SMT. For more complex operations that possibly read values from memory such as BigInt with bit-width larger than 64, we simply encode them as UF. This may introduce false alarms if TurboFan optimizes the operations to constants for inputs not appearing in the if-then-else expression. We assume that all function calls do not update the memory. This may introduce false positives and negatives.

5.3.2 Nondeterminism. It is known that JS may exhibit nondeterminism when the NaN (Not-a-Number) value is involved [4]. There are multiple bit-representations of NaN, and a JS engine can pick any of the NaN bit representations. We approximate NaN handling by removing the nondeterminism and considering all NaN values as equal. This is beneficial for two reasons. First, showing the compiler correctness of a program having nondeterministic behavior is expensive in SMT because the refinement relation between two behaviors becomes a subset relation rather than simple equality. This causes using ∀ quantifiers. Second, FP theory in SMT solver does not distinguish NaN of different bit representations. This approximation facilitates using the FP theory without additional costs.

5.3.3 Internal Data Structures of V8. TurboFan's typical IR programs contain pointers to TurboFan's internal data structures in memory. Faithfully encoding the memory layout of these data structures is essential for successful validation because, without the layout knowledge, UB Checker may report false alarms. However, it is impractical to fully define their layouts for two reasons. First, V8 has a lot of internal data structures whose layouts depend on the target architecture. Second, the data structures continuously change as V8 evolves.

To reduce false alarms from this issue, we split the memory into two regions: AngelicMemory and DemonicMemory. AngelicMemory contains a set of memory blocks assumed to be pre-allocated by V8. We assume that any operation on the pointer to AngelicMemory as well as its transitive users, cannot raise undefined behavior (e.g.,

pointer dereference always succeeds). This may introduce missing bugs but removes false alarms. Conversely, DemonicMemory is a memory region not pre-allocated by V8. `Load` or `Store` to a memory block in this region checks out-of-bounds as usual.

For the input parameters, we put the memory block referred by the TaggedPointer into AngelicMemory. This helps our encoding become more robust across the architectures and avoid the false alarms related to the V8's various internals. In TurboFan IR, there is a type-check operator used to ensure the kind of referred object before accessing it. When this check fails, the program is deoptimized. This is a speculative guard preventing a program from illegal memory access. If we put the parameter referred block into DemonicMemory, we should encode this operator thoroughly. Otherwise, our UB Checker may suffer from the false positive that is unreachable indeed or build the wrong deoptimization condition which increases the false negative. However, it is impractical to encode this type-check operator thoroughly since many kinds of objects exist in V8. Thereby, we choose to put the parameter referred block into AngelicMemory.

6 CROSS-LANGUAGE TV

In this section, we describe our efforts to extend TurboTV to support TV across different languages. We combine TurboTV with Alive2, an SMT-based translation validator for LLVM [23, 24], and validate translations from LLVM IRs to TurboFan IRs. The idea is to use Wasm [38] as an intermediate language as LLVM has a Wasm backend and TurboFan has a Wasm frontend. Thus, we check the refinement relation between an LLVM source function and its TurboFan target by simply combining the two tools. Since LLVM and TurboFan have different memory models, we focus on functions whose parameters and return values are numeric values.

Given an LLVM IR function, we encode the semantics of the source function and its TurboFan target as SMT formulas. We first generate the Wasm code of the function using LLVM and encode the semantics of LLVM IR as an SMT formula using Alive2. Then, we translate the Wasm code to TurboFan IR using TurboFan and encode the semantics as an SMT formula using TurboTV. Finally, we check the refinement between the two programs via the derived SMT formulas using the refinement formula generated by Alive2.

According to the specification [39], Wasm programs also do not have UB, so TurboFan IR should not have UB either. We use the UB Checker to inspect that the TurboFan IR function does not raise UB as in the JS case. However, unlike Wasm, LLVM IR may have undefined behavior. Therefore, we cannot use the EQ Checker that only checks equivalence but not refinement. With this cross-language TV, we found a bug in the Wasm backend of LLVM. The details of the bug will be discussed in the evaluation section.

7 COMBINING FUZZING AND TV

This section explains how we combine fuzzing and TurboTV. Since TurboFan compiles JS code at runtime, it is impractical to use TurboTV during the application execution for the overhead. Thus, we employ fuzzing to generate a large number of JS functions and check the correctness of TurboFan for the corpus using TurboTV. This combination enables us to discover latent bugs that are not observable as outcomes (e.g., crashes) by fuzzing alone while generating functions to trigger various optimization passes effectively. We also introduce our effort to test TurboTV itself using a fuzzer.

Validation Corpus Generation. We first generate a large number of JS programs as validation corpus. This process is designed to effectively trigger diverse optimization passes in V8. Therefore, it is crucial to generate not only a variety of JS statements in a function body but also different calling contexts for function specialization. The algorithm consists of two phases: 1) function body generation and 2) call augmentation.

The first phase follows the standard process of fuzzing. We establish an initial set of seed programs using generation-based fuzzing. Given a time budget, the fuzzer repeatedly generates random JS functions called with a default argument (e.g., `0`) and collects them if they have any gain in terms of coverage of V8. Then, we generate more programs using mutation-based fuzzing from the initial seed programs in a similar way. We used Fuzzilli [32], a state-of-the-art JS fuzzer for this process.

Furthermore, we specialized Fuzzilli to efficiently generate JS functions for TurboTV. Fuzzilli is primarily designed to test the overall pipeline of JS engines (e.g., parser or interpreter) rather than being specialized for optimization. Therefore, the vanilla version usually generates programs that do not trigger various optimizations. We implemented two key modifications to Fuzzilli to improve efficiency. First, we configured Fuzzilli to generate code within the scope of TurboTV. Fuzzilli provides a set of *generators*, each of which generates specific types of statements or expressions. For TurboTV, we only turned on the generators for our validation scope. Second, we modified Fuzzilli to actively use function parameters in the body. JIT compilers often specialize functions based on the types or values of their arguments. However, we observed that the vanilla fuzzer often generates programs that do not use the parameters at all. So we made Fuzzilli randomly change the variables used in the body to the parameters.

The second phase derives different calling contexts for the functions generated from the first phase. The algorithm is parameterized with a set of constant values in JS. The set is used to generate programs that use different constant values as arguments. In our experiments, we chose 14 constants, each of which is a representative value of a type in JS (e.g., `0`, `[]`, and `undef`). Since calling the same function multiple times with different arguments affects the optimization passes in V8, we augment the generated functions differently. In our experiments, we append two calls for each function with all combinations of the chosen constant set. For example, two function calls `f(0); f([]);` can be added to a generated function `f`. Note that this augmentation only has marginal overhead since it just enumerates candidate arguments without execution.

Using TurboTV as Fuzzing Oracle. We further incorporate TurboTV into the fuzzing process by using it as a fuzzing oracle. Existing fuzzers typically run the generated programs with the engine and observe the outcomes, such as crashes or differences between engines. Instead, whenever the fuzzer generates a JS function that achieves new code coverage in TurboFan, we validate its JIT compilation using TurboTV. By doing so, TurboTV improves the detectability of fuzzers while maintaining their efficiency in generating new JS functions. TurboTV can discover latent miscompilations

during intermediate optimization steps and also consider all possible input values of compiled functions. Moreover, we demonstrate that the cost of TV is amortized to a small fraction of the running time when the fuzzer runs long enough.

Testing TurboTV. We also utilized fuzzing to check the correctness of TurboTV itself, following a similar approach to previous work on compiler [22]. The idea is to use fuzzer to generate a pair of JS functions that are semantically *different* and check whether TurboTV incorrectly validates them as *equivalent*. We first generate a large number of random JS functions using Fuzzilli. Next, we run the functions with fixed arguments and collect the return values. We used 0 and 1 for the arguments. Then, we partition the collected functions into equivalent classes based on the return values. Finally, we derive a pair of functions from different equivalent classes and check whether TurboTV incorrectly validates them as equivalent.

8 EVALUATION

Our evaluation aims to answer the following research questions:

RQ1 How effective is TurboTV to validate JS JIT compilations?
RQ2 How effective is the cross-language TV?
RQ3 How effective is TurboTV as fuzzing oracle?

We implemented TurboTV comprising 12K lines of OCaml code for encoding the semantics of 306 out of 914 operators of TurboFan in V8 11.7.2, including arithmetic, bitwise, string, memory, and control operators. We instrumented TurboFan to extract the IR of each optimization pass. We used Z3 [?] as the SMT solver and implemented the fuzzer on top of Fuzzilli [32]. Our experiments are conducted on a Linux machine with Intel Xeon 2.90GHz. We evaluated TurboTV on the following four benchmarks:

- Bug: We collected 9 optimization bugs of TurboFan from the Chromium bug tracker [6] reported between Jan 2020 and Dec 2022. We excluded other bugs reported before because V8 fundamentally changed the memory model in Dec 2019.
- UnitJS: We collected 580 loop-free JS functions for testing the JIT compiler in mjsunit, the regression test suite of V8 [19].
- Corpus: We generated 196,000 JS programs using our validation corpus generator. They are augmented with two function calls with 14 constants from 1,000 initial loop-free JS functions.
- UnitLLVM: We collected 3,580 LLVM IR functions in the regression test cases for LLVM where the parameter and return types are integer or float. We excluded functions that are not correctly compiled to Wasm by LLVM.

8.1 Precision and Scalability of TurboTV

We first evaluate the effectiveness of TurboTV in discovering previously reported bugs in TurboFan. Table 1 shows the list of bugs in the Bug benchmark and the validation results. For each bug, we ran TurboTV for the JS function attached to the bug report and validated whether the JIT compilation is correct. We set the timeout to 3 minutes for each validation of reduction and IR.

The results indicate that TurboTV is robust enough to discover real bugs in TurboFan. Our encoding covers a large portion of instructions in TurboFan and does not miss any real bugs in the benchmark. In total, there are 298 IRs and 114 reductions extracted from TurboFan. Among them, 183 (61%) IRs and 85 (75%) reductions consist of instructions supported by TurboTV. Among the bugs, three can be detected by the UB Checker, and the EQ Checker detects the remaining six. Overall, TurboTV does not report any false positives and only results in 20 timeouts.

Table 1: Effectiveness of TurboTV in discovering known bugs. IR_{All} (resp., Rdc_{All}), and IR_{TV} (resp., Rdc_{TV}) are the number of unique IRs (resp., reductions) extracted from TurboFan and supported by TurboTV, respectively. FP and TO are the numbers of false positives and timeout. Detect indicates the checker that detects the bug. The EQ checking is meaningless if the bug is detected by the UB Checker.

Bug ID	UB Checker				EQ Checker				Detect
	IR_{All}	IR_{TV}	FP	TO	Rdc_{All}	Rdc_{TV}	FP	TO	
1195650 [8]	42	13	0	0	-	-	-	-	UB
1198705 [9]	63	32	0	8	-	-	-	-	UB
1404607 [15]	58	33	0	3	-	-	-	-	UB
1126249 [7]	20	20	0	0	17	17	0	0	EQ
1199345 [10]	13	13	0	0	10	10	0	0	EQ
1200490 [11]	41	30	0	0	37	26	0	0	EQ
1234764 [12]	19	19	0	2	15	15	0	2	EQ
1234770 [13]	12	12	0	3	9	9	0	2	EQ
1323114 [14]	30	11	0	0	26	8	0	0	EQ
Total	298	183	0	16	114	85	0	4	

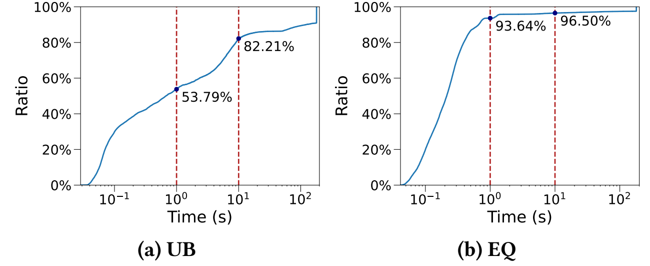


Figure 6: Cumulative distribution of the validation time on the Corpus benchmark. The x-axis is in the log scale.

Next, we evaluate the performance of TurboTV on a large set of JS programs: the UnitJS and Corpus benchmarks. Table 2 shows the results. Among 4,935 targeted IRs and 4,387 targeted reductions in the UnitJS benchmark, both UB and EQ Checker showed a significantly low false positive ratio (1%). Similarly, the result on the Corpus benchmark, which contains larger JS files than UnitJS, demonstrates the high accuracy of TurboTV. For the UB Checker, TurboTV still produces less than 2% of false alarms. Notice that there is no false positive reported by the EQ Checker. One of the main reasons for false positives is out-of-scope objects. If the two IRs return an object that TurboTV does not support, TurboTV cannot check their equality but soundly alarms such cases.

Moreover, we measured the validation time of TurboTV. Fig. 6(a) and 6(b) show the cumulative distribution of the validation time on the Corpus benchmark. The results show that most validations are completed within 10 seconds. Especially for the EQ check, over 96% of the validations are completed within 10 seconds. This also indicates that 10 seconds can be a reasonable time budget when TurboTV is used as a fuzzing oracle.

Table 2: Effectiveness of TurboTV for large benchmarks. JS/LLVM indicate the number of JS and LLVM programs. IR_{All} (resp., Rdc_{All}), and IR_{TV} (resp., Rdc_{TV}) are the number of unique IRs (resp., reductions) extracted from TurboFan/LLVM and supported by TurboTV, respectively. Val indicates the number of validated IRs and reductions validated by TurboTV. TP and Time show the number of true positives and the average time for each validation except for timeouts. For cross-language TV, we use the Refinement checker (Sec. 6) rather than the EQ checker.

Benchmark	JS/LLVM	UB Checker						EQ / Refinement Checker						
		IR_{All}	IR_{TV}	Val	FP	TO	Time	Rdc_{All}	Rdc_{TV}	Val	TP	FP	TO	Time
UnitJS	580	16,425	4,935	4,434	32	469	9.42s	16,011	4,387	4,018	0	41	328	2.61s
Corpus	196K	172,921	14,974	13,402	251	1,321	9.16s	160,324	13,870	13,572	0	0	298	0.71s
UnitLLVM	3,580	3,580	2,659	2,659	0	0	0.11s	3,580	2,659	2,498	90	10	61	0.70s

According to the results, TurboTV is scalable to validate the compilation of a large set of JS programs with low false positive rates. This fact indicates that TurboTV can be used to effectively check the correctness of JIT compilers with a large corpus in practice.

8.2 Effectiveness of Cross-Language TV

We evaluate the effectiveness of cross-language TV in UnitLLVM. For each IR, we performed cross-language TV to validate whether a function of LLVM and TurboFan are semantically equivalent. We set the timeout to 3 minutes for each validation of IR.

Table 2 shows the result. Among 3,580 LLVM IRs in UnitLLVM, 2,659 IRs are supported by TurboTV. The UB Checker successfully validated all IRs. The validations take 0.11 seconds on average. The Refinement Checker discovered 90 miscompilations and showed a significantly low false positive ratio (0.4%). The main reason for the false positives is due to the physical memory models of LLVM and TurboFan that are not completely captured by Alive2 and TurboTV. Such false positives can happen when IRs contain instructions using physical addresses as values such as `ptrtoint`.

The 90 miscompilations are due to one common root cause. In LLVM, an integer function parameter tagged with `signext` must be lowered to a larger machine register containing its sign-extended value. Either caller or callee is responsible for storing sign-extended values, but LLVM's Wasm backend sometimes did not insert sign extensions in any place.

Fig. 7 shows a representative case of the bug. In the LLVM source IR, the function `foo` casts the first argument `%x` into a 32-bit integer `%e`, and returns `%y - %e`. When LLVM compiles this function (`foo`) to Wasm code using the `-O2` option, it converts the first parameter (`%x`) to a 32-bit integer due to the absence of a 1-bit integer type in Wasm. Additionally, LLVM assumes that the first argument has been sign-extended by the caller because the parameter `%x` possesses the `signext` attribute.

For example, if another function `main` calls function `foo` with values 1 and 0, the compiled Wasm function should call function `foo` with values -1 and 0. Note that the sign extension of a 1-bit integer 1 to a 32-bit integer is -1. However, when LLVM compiles the `main` function with option `-O0`, the compiled function incorrectly calls function `foo` with arguments 1 and 0. This makes the function `foo` in the source and target return different values.

Initially, we decided that this was a bug in `-O2` compilation of `foo`. Since Wasm ABI [37] has multiple versions, we decided to encode the Wasm calling convention based on LLVM's `-O0` option. Also, the description of `signext` of the LLVM Language Reference

```
define i32 @foo(i1 signext %x, i32 %y) {
  %e = zext i1 %x to i32
  %r = sub i32 %y, %e
  ret i32 %r
}
```

(a) Source LLVM IR.

```
func $foo (param i32 i32) (result i32)
  local.get 1
  local.get 0
  i32.add
```

(b) Target Wasm code.

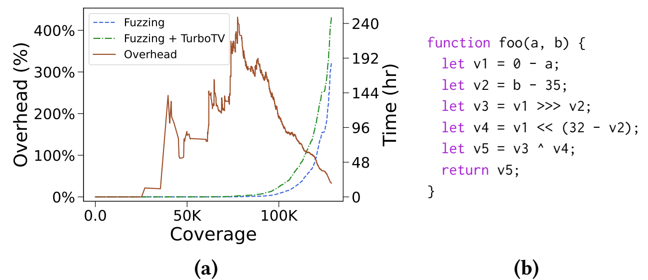
```
declare i32 @foo(i1 signext, i32)

define i32 @main() #0 {
   %0 = call i32 @foo(i1 1, i32 0)
   ret i32 %0
}
```

(c) LLVM IR calling function foo.

```
func $main (param i32 i32) (result i32)
  i32.const 1
  i32.const 0
  call foo
  return
```

(d) Miscompiled Wasm code.

Figure 7: Miscompilation found in the LLVM Wasm backend.



(a) (b)

Figure 8: Overhead of Fuzzilli combined with TurboTV (a) and JS code snippet that triggers a miscompilation (b).

manual left the bit-width to extend as determined by the target machine. This naturally made the `-O2` optimization of `foo` classified as wrong by TurboTV. However, after a discussion with LLVM developers[3], it was concluded that the translation of `main` with `-O0` was problematic. A patch that fixes this bug was reviewed by developers and merged to LLVM.

This result indicates that our cross-language TV is effective in finding real bugs in practice. The bugs cannot be discovered by Alive2 or TurboTV alone since the tools only validate transformations to the same IR. However, the combination can effectively validate the LLVM backend and the TurboFan frontend.

8.3 Effectiveness of TurboTV as Fuzzing Oracle

To demonstrate the scalability of TurboTV,we measured the overhead of validator invocations by comparing the running time of

[3] https://github.com/llvm/llvm-project/issues/63388

TurboTV with the running time of its fuzzer only. We ran the fuzzing algorithm described in Sec. 7 for 7 days using a typical fuzzing oracle. That is, we used V8 as an oracle and checked whether the oracle produced crashes. Then, we measured the overhead of our combination to achieve the same edge coverage of TurboFan by using TurboTV as the fuzzing oracle. Since most validations are completed within 10 seconds, as shown in Fig. 6, we set the timeout to 10 seconds for each validation.

Fig. 8(a) shows the running time and overhead. The dotted lines depict the accumulated time taken for the fuzzing process with and without using TurboTV to achieve the same coverage. The solid line represents the overhead at each point. Overall, the overhead increases only for the first 86 minutes until covering 77K edges. After that, it dramatically decreases and finally becomes 36%. In total, the combination took 229 hours to cover 127K edges, while the baseline fuzzing took 168 hours. Notice that we sequentially ran TurboTV and the fuzzer; the fuzzer generates the next JS function after the validation of the previously generated one. Since the two tools run independently, we can run them in parallel to reduce the overhead further.

We also demonstrate the effectiveness of the combination in terms of detecting known bugs listed in Table 1. For this purpose, we first ran state-of-the-art fuzzers, Fuzzilli and FuzzJIT [36] for 7 days. but failed to generate JS functions that trigger these known bugs. Thus, we restricted Fuzzilli to only use the operators that are necessary to trigger the bugs and a limited set of constants. Then we evaluated its effectiveness when combined with TurboTV.

Combined with the restricted fuzzer, TurboTV successfully detected Issue 1234764 [12] within only 15 minutes, whereas the fuzzer alone failed to reproduce the bugs within 24 hours. Fig. 8(b) shows a JS function that reveals the bug. Note that the function triggers the bug only when a specific value is used for the parameter b (e.g., 3). The state-of-the-art fuzzers only use a fixed set of values for function parameters, so they are unlikely to trigger the bug. However, TurboTV considers all possible values for the parameter by symbolically encoding the function semantics and consequently detects the bug without choosing a specific parameter value.

The results indicate that TurboTV can be effectively used as a precise fuzzing oracle for JIT compiler testing. As the coverage increases, fuzzers typically have a hard time generating test cases to cover new code paths. Thus, TurboTV can complement the fuzzers by precisely checking the JIT compilation of the generated test cases and amortizing overheads as the coverage increases.

9 RELATED WORK

Recent years have witnessed an increasing interest in translation validation, but all the previous work targets AOT compilers [1, 23, 24, 33]. We present new ideas for effectively applying translation validation for a JIT compiler. We designed a staged strategy by exploiting the characteristics of JS and SMT encoding for TurboFan IR semantics considering deoptimization.

There have been several works on verified JIT compilers. Vera [5] rewrote the range analysis module of the JIT compiler in Firefox and formally verified it using an SMT solver. Barrière et al. also presented a formally verified JIT compiler using Coq [2, 3]. These approaches guarantee the full correctness of (a part of) compilers, whereas TurboTV only validates a particular compilation. Instead, TurboTV is a cheaper solution for checking the correctness of an existing industrial JIT compiler without reimplementing it.

Our formal semantics is inspired by previous work that formalizes the denotational semantics for the value operators as well as operational semantics for the control flow operators of SoN [17]. Based on their work, we define the formal semantics of TurboFan IR, one of the most popular applications of SoN IR, and devise its SMT encoding for translation validation.

There is a large body of research on testing JS engines and JIT compilers. Fuzzing techniques have been successfully applied to test JS engines [20, 26, 30, 32]. They randomly generate JS programs and try to find crashes in the engines. Recent differential testing techniques detect non-crashing bugs [4, 30, 36]. They crosscheck the behavior of the same JS program executed by different interpreters or JIT compilers. However, bugs in JS engines are not always externally observable for all inputs. TurboTV checks the correctness of a compilation for all input and discovers miscompilation bugs regardless of the runtime execution path. We also demonstrated that TurboTV can be effectively combined with existing fuzzers.

10 DISCUSSION

In the future, we will add support for more operators. We encoded 61% and 54% of the operators in the Common and Simplified categories. In the Machine category, we only focused on x86 and encoded 52.5% of the x86 operators. Supporting the remaining operators will be straightforward. On the other hand, we did not encode the semantics of JS operators because they are not related to most of the observed bugs. Since JS operators are complicated, it may be challenging to design efficient SMT encoding.

We plan to extend TurboTV to support interprocedural optimizations and functions containing loops. Validating interprocedural optimizations requires knowing the semantics of invoked functions. If the functions are subsequently JIT compiled by TurboFan, TurboTV can encode their semantics, making the validation possible. To support loops, TurboTV needs to synthesize loop invariants, which can be found using existing techniques [34]. If loops are known to iterate at most constant times, we can simply unroll the loop and validate the transformed programs.

11 CONCLUSION

We proposed TurboTV, an SMT-based TV for TurboFan. We presented a staged strategy for TV for JS that enables us to derive simpler SMT queries than the conventional approach. Also, we demonstrated that TurboTV can be effectively combined with fuzzing. We generated a large corpus of JS functions and used it to check the correctness of TurboFan. Lastly, we applied TurboTV to cross-language TV between LLVM and TurboFan using Wasm as an intermediate language. The combination discovered a new miscompilation of the LLVM Wasm backend.

ACKNOWLEDGMENTS

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1A5A1021944, 2021R1C1C1003876).

REFERENCES

[1] Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. 2022. SMT-Based Translation Validation for Machine Learning Compiler. In *International Conference on Computer Aided Verification (CAV)*. Springer.

[2] Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. 2021. Formally Verified Speculation and Deoptimization in a JIT Compiler. *Proceedings of the ACM on Programming Languages* 5, POPL (2021).

[3] Aurèle Barrière, Sandrine Blazy, and David Pichardie. 2023. Formally Verified Native Code Generation in an Effectful JIT: Turning the CompCert Backend into a Formally Verified JIT Compiler. *Proceedings of the ACM on Programming Languages* 7, POPL (2023).

[4] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. Jit-Picking: Differential Fuzzing of JavaScript Engines. In *ACM Conference on Computer and Communications Security (CCS)*. ACM.

[5] Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Towards a Verified Range Analysis for JavaScript JITs. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[6] Chromium. 2016. Chromium Bug Tracker. https://bugs.chromium.org/p/chromium/issues/list

[7] Chromium. 2020. Issue 1126249. https://bugs.chromium.org/p/chromium/issues/detail?id=1126249

[8] Chromium. 2021. Issue 1195650. https://bugs.chromium.org/p/chromium/issues/detail?id=1195650

[9] Chromium. 2021. Issue 1198705. https://bugs.chromium.org/p/chromium/issues/detail?id=1198705

[10] Chromium. 2021. Issue 1199345. https://bugs.chromium.org/p/chromium/issues/detail?id=1199345

[11] Chromium. 2021. Issue 1200490. https://bugs.chromium.org/p/chromium/issues/detail?id=1200490

[12] Chromium. 2021. Issue 1234764. https://bugs.chromium.org/p/chromium/issues/detail?id=1234764

[13] Chromium. 2021. Issue 1234770. https://bugs.chromium.org/p/chromium/issues/detail?id=1234770

[14] Chromium. 2022. Issue 1323114. https://bugs.chromium.org/p/chromium/issues/detail?id=1323114

[15] Chromium. 2023. Issue 1404607. https://bugs.chromium.org/p/chromium/issues/detail?id=1404607

[16] Cliff Click and Keith D. Cooper. 1995. Combining Analysis, Combining Optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17, 2 (1995), 181–196.

[17] Delphine Demange, Yon Fernández de Retana, and David Pichardie. 2018. Semantic Reasoning about the Sea of Nodes. In *International Conference on Compiler Construction (CC)*. ACM.

[18] ECMA International. 2022. ECMA-262 - ECMAScript Language Specification. https://www.ecma-international.org/publications-and-standards/standards/ecma-262/

[19] Google. 2008. V8. https://v8.dev

[20] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[21] Igor Sheludko and Santiago Aboy Solanes. 2020. Pointer Compression. https://v8.dev/blog/pointer-compression

[22] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *ACM SIGPLAN International Conference on ProgrammingLanguage Design and Implementation (PLDI)*.

[23] Juneyoung Lee, Dongjoo Kim, Chung-Kil Hur, and Nuno P Lopes. 2021. An SMT Encoding of LLVM's Memory Model for Bounded Translation Validation. In *International Conference on Computer Aided Verification (CAV)*.

[24] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *ACM SIGPLAN International Conference on ProgrammingLanguage Design and Implementation (PLDI)*. ACM.

[25] Meta. 2022. React Native. https://reactnative.dev

[26] Mozilla Security. 2007. Funfuzz. https://github.com/MozillaSecurity/funfuzz

[27] OpenJS Foundation. 2022. Electron. https://www.electronjs.org

[28] OpenJS Foundation. 2022. JerryScript. https://jerryscript.net

[29] OpenJS Foundation. 2022. NodeJS. https://nodejs.org

[30] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *IEEE Symposium on Security and Privacy (SP)*. IEEE.

[31] A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation Validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[32] Samuel Groß. 2022. Fuzzilli. https://github.com/googleprojectzero/fuzzilli

[33] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[34] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2Inv: A Deep Learning Framework for Program Verification. In *International Conference on Computer Aided Verification (CAV)*.

[35] The Hybrid Group. 2022. Cylon.Js. https://cylonjs.com

[36] Junjie Wang, Xiaoning Du Zhiyi Zhang, Shuang Liu, and Junjie Chen. 2023. FuzzJIT: Oracle-enhanced Fuzzing for JavaScript Engine JIT Compiler. In *USENIX Security Symposium (Security)*. USENIX Association.

[37] World Wide Web Consortium. 2022. Webassembly ABIs. https://www.webassembly.guide/webassembly-guide/webassembly/wasm-abis

[38] World Wide Web Consortium. 2023. Webassembly Organization Web Page. https://webassembly.org/

[39] World Wide Web Consortium. 2023. WebAssembly Specification. https://webassembly.github.io/spec/core/