Translation Validation for JIT Compiler in the V8 JavaScript Engine

1 PURPOSE

This document describes the artifact for TurboTV presented in the paper "Translation Validation for JIT Compiler in the V8 JavaScript Engine". TurboTV is a translation validator for the JavaScript (JS) just-in-time (JIT) compiler of V8. This artifact provides an environment to run TurboTV to validate TurboFan's (JIT compiler of V8) optimization passes. We claim the available and reusable badges for our artifact.

2 PROVENANCE

The artifact is provided as a Docker image that includes the experiment environment, the source code of the tool, and the scripts to run the tool and evaluate the results. We publicize the artifact on Zenodo (https://doi.org/10.5281/zenodo.10453785). The Zenodo archive includes the artifact, the LICENSE, and a copy of the paper. Our artifact does not have any security, privacy, or ethical concerns.

3 SETUP

In this section, we describe how to set up the artifact for TurboTV.

3.1 Hardware and Software Requirements

We ran the experiment on the machines equipped with Intel(R) Xeon(R) Gold 6226R CPU (2.90GHz) with 64 cores and 512 GB of RAM. It is possible to run the experiment on a machine with fewer CPU cores and smaller RAM, but it is recommended to use the artifact with more than 4 CPU cores and 16 GB of RAM.

We provide our artifact as a Docker image based on Ubuntu 22.04 and Docker 24.0.7. Since the volume of the Docker image is 23 GB, we recommend at least 30 GB of disk space.

3.2 Installation

(1) Download the `turbo-tv.tar.gz` from the Zenodo website.
(2) Then, load and run the Docker image from the file.

```
docker load < turbo-tv.tar.gz
docker run -it --privileged prosyslab/turbo-tv
```

4 BASIC USAGE

This section describes the utilization of TurboTV and the subsequent analysis of outputs. Essentially, TurboTV extracts Intermediate Representations (IRs) during JIT compilation and performs translation validation process. The process divided into two components: the Undefined Behavior (UB) Checker and the Equivalence (EQ) Checker, as detailed in Section 2.3.2 of the paper. The UB Checker checks the presence of undefined behavior within a single IR, while the EQ Checker checks the semantic equivalence between two IRs before and after reduction. We provide a script designed to assist TurboTV in receiving JS files and executing Translation Validation. The detailed usage of the script is as follows:

(1) Select a version of V8 for a specific bug. For example, the following command selects the version of V8 for the bug 1199345.

```
./exp v8 --select --issue 1199345
```

(2) Run TurboTV on a JS file. For example, the following command runs the UB Checker on the JS file `1199345.js`.

```
./exp turbo-tv --check-ub issues/1199345/1199345.js
```

The following command runs the EQ Checker on the same file.

```
./exp turbo-tv --check-eq issues/1199345/1199345.js
```

Once a validation fails, TurboTV presents a counterexample that describes a potential miscompilation. For example, the above EQ check generates the following output:

```
========[Check EQ of js(s) in target dir]========
. . .
/home/user/turbo-tv-exp/workbench/1199345/2: X
c.e. =>
/home/user/turbo-tv-exp/workbench/1199345/2.ce
. . .
```

The output indicates that the counterexample is stored at `/home/user/turbo-tv-exp/workbench/1199345/2.ce`. The provided counterexample will be as follows:

```
Result: Not Verified
CounterExample:
Parameters:
Parameter[0]: TaggedSigned(0)
. . .
State of src
#0:NumberConstant(0) [Range (0.0000, 0.0000)] =>
Value: TaggedSigned(0)
. . .
```

The provided counterexample illustrates a potential miscompilation when the first parameter (i.e., `Parameter[0]`) of the function in `1199345.js` is constant `0` (i.e., `TaggedSigned(0)`).

5 REPRODUCE EVALUATION

In this section, we describe the process for reproducing the experiments conducted in our paper. We provide scripts to easily replicate the experiment for each research question (RQs). Certain experiments necessitate several hours to complete. For convenience, all experimental results are already available at `/home/user/turbo-tv-exp/eval`.

Note that some of the experiments may produce different results from those in the paper. All experiments in the paper were conducted in the environment described in Sec 3.1 and concurrently conducted using 64 cores. Thus, the experimental results may vary depending on the reviewer's environment. For convenience, we provide estimated execution times using 64-core and 4-core environments, for each experiment.

5.1 Precision and Scalability of TurboTV (RQ 1)

5.1.1 Effectiveness of TurboTV in discovering known bugs. (Table 1). The experiment evaluates the precision of TurboTV by discovering previously reported bugs in TurboFan. To replicate the experiment, execute the following command. (~30 minutes using 64 cores, ~2 hours using 4 cores)

```
./exp eval --precision
```

The output will be as follows:

```
======issue 1126249 UB check result.======
IR_ALL 20
IR_TV 20
ALARM(s) 0
TIMEOUT 3
Counter examples are saved to
"./eval/precision-1126249-ub-ce.txt"
======issue 1126249 EQ check result.======
RDC_ALL 17
RDC_TV 17
ALARM(s) 1
TIMEOUT 0
Counter examples are saved to
"./eval/precision-1126249-eq-ce.txt"
. . .
```

The output describes the results of both checkers for each known bug and corresponds to the rows of Table 1 in the paper. Specifically, **IR_ALL** and **RDC_ALL** indicate the number of extracted IRs or reductions, **IR_TV** and **RDC_TV** indicate the number of IRs and reductions supported by TurboTV, **ALARM(s)** indicates the number of IRs and reductions that are not verified by each checker. Note that **ALARM(s)** includes both true positives (TP) and false positives (FP). Once a validation fails, the counterexample is saved in the path specified in the output.

5.1.2 Effectiveness of TurboTV for a large set of JS programs. (Table 2). The experiment evaluates the performance of TurboTV. Since this experiment is conducted on a large set of JS programs, it can take a long time to complete. To replicate the experiment, run the following command (~3 hour with 64 cores, 8 hours with 4 cores):

```
./exp eval --scalability
```

For convenience, we also provide a scaled-down version of the experiment. We randomly selected 5 and 10 samples from the UnitJS and Corpus benchmarks, respectively. The following command replicates the experiment on these small benchmarks (~5 minutes with 64 cores, 10 hours with 4 cores):

```
./exp eval --scalability --small
```

The output provides the summary of the translation validation results for the benchmarks. **Total Elapsed** denotes the cumulative time spent on the checking process, while **Avg Elapsed** indicates the average time consumed per IR. The other part of the output is the same as in the previous section.

5.2 Effectiveness of Cross-Language TV (RQ 2)

The experiment evaluates the effectiveness of cross-language TV on the LLVM unit test cases. To replicate the experiment, run following command (~30 minutes with 64 cores, 1 hours with 4 cores):

```
./exp eval --cross-validation
```

The output format is as shown in Sec 5.1.2, wherein 100 alarms are generated in the Refinement check. Among these alarms, there are 90 true positives (TPs) relating to the bugs documented in the paper, and 10 false positives (FPs).

For convenience, we provide a command to run cross-language TV on a single LLVM IR. For example, the following command reproduces the bugs in the paper.

```
./exp v8 --select
./exp turbo-tv --cross-refine benchmarks/unit-ll/bool-ext-inc.ll
```

The above Refinement check generates the following output:

```
========[Validate ll(s) in target dir]========
. . .
./workbench/bool-ext-inc/bool-ext-inc.ll: X
c.e. =>
./workbench/bool-ext-inc/bool-ext-inc.ce
. . .
```

The output indicates that the counterexample is stored at `./workbench/bool-ext-inc/bool-ext-inc.ce`.
The provided counterexample will be as follows:

```
sat . . .
(define-fun p1 () (_ BitVec 69)
#b. . . 00000)
(define-fun p0 () (_ BitVec 69)
#b. . . 00001)
. . .
```

The provided counterexample illustrates a potential problem when the first and second parameters (i.e., `p0`, `p1`) of the function in `bool-ext-inc.ll` are constant 1 and `0` (i.e., `#b. . . 00001`, `#b. . . 00000`).

5.3 Effectiveness of TurboTV as Fuzzing Oracle (RQ 3)

This experiment evaluates the effectiveness of TurboTV as a fuzzing oracle. Through a fuzzing process, we generate random JS files and measure an overhead for TurboTV to validate these JS files. The overhead denotes the ratio between the time consumed by TurboTV and the fuzzer.

Since the corpus generation process is non-deterministic and time-consuming, we provide the corpus used in the paper. The following command will validate the pre-generated corpus (~12 hours with 64 cores, 24 hours with 4 cores):

```
./exp eval --overhead --corpus benchmarks/corpus-overhead
```

For convenience, we provide a command to generate a corpus from scratch. For example, the following command generates random JS files for 10 seconds and measures overhead on the generated corpus (~3 minutes with 64 cores, 5 minutes with 4 cores):

```
./exp eval --overhead --timeout 10
```

The generated corpus is saved in `./fuzz-out/corpus`. The output of above command will be as follows.

```
. . .
Total Overhead: 171.86%
=> The time spent on each JS is saved at ./overhead.json
. . .
```

The output describes the overhead (`Total Overhead`) incurred when employing TurboTV as a fuzzing oracle for the corpus. The time spent validating each JS file in the corpus is stored in the JSON file. Note that translation validations are performed concurrently, so the overhead may rise if fewer cores are used than in our environment. For instance, if validating a single IR takes 0.1 seconds, the validation of 30 IRs with 30 cores takes 0.1 seconds. but with only one core, it takes 3 seconds.