USENIX'23 Artifact Appendix: <FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler>

Junjie Wang[1*], Zhiyi Zhang[2*], Shuang Liu[1+], Xiaoning Du[3], and Junjie Chen[1]

[1]College of Intelligence and Computing, Tianjin University
[2]CodeSafe Team, Qi An Xin Group Corp.
[3]Monash University

A Artifact Appendix

This artifact appendix is meant to be a self-contained document which describes a roadmap for the evaluation of FuzzJIT.

A.1 Abstract

FuzzJIT is a fuzzing tool for JavaScript engines JIT compiler, built on top of Fuzzilli [1]. FuzzJIT maintains a queue that contains all samples that triggered new code coverage in the testing subjects. At the start of each fuzzing round, FuzzJIT selects a test case from the queue and mutates it to generate new test cases. Generated new test cases are executed, and the test cases that triggered new code coverage are then added to the fuzzing queue for further mutation. Our main contributions include a for-loop structure to trigger the JIT compilers, a test function embedding JIT compiler bugs revealing elements, and an enhanced oracle to check if the test function output differently before/after the JIT compilation.

A.2 Description & Requirements

In this section, we list the information necessary to recreate the same experimental setup we have used to run our artifact. We also list the hardware and software requirements to run our artifact. At last, we list benchmarks used to produce results with our artifact.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

FuzzJIT can be obtained from GitHub: https://github.com/SpaceNaN/fuzzjit/commit/a3d3f6da7f7f8577476892d6135eee6c50afc7ad.

A.2.3 Hardware dependencies

In our experiments, we used a workstation with a processor of 12th Gen Intel Core i9-12900K*24 and 32 GB memory. Any similar configuration should work too.

A.2.4 Software dependencies

In our experiment, FuzzJIT runs on a 64-bit Ubuntu 22.04.01 LTS system. Other Linux operating systems should work too. The same as Fuzzilli, FuzzJIT is written in Swift, therefore, the installation of Swift is required. Swift 5.7 and 5.3 are tested working.

A.2.5 Benchmarks

Our testing subjects include four mainstream JavaScript engines, JavaScriptCore (the JavaScript engine of the Safari browser), V8 (the JavaScript engine of the Chrome browser), Spidermonkey (the JavaScript engine of Firefox), and ChakraCore (the JavaScript engine of Edge). In our evaluation, we compared FuzzJIT with four baselines, including Jsfunfuzz [3], Superion [4], DIE [2], and Fuzzilli [3].

A.3 Set-up

In this section, we list the installation and configuration steps required to prepare the environment to be used for the evaluation of our artifact.

A.3.1 Installation

The running procedure of FuzzJIT is the same with Fuzzilli.

1. Download Swift from its download page: https://www.swift.org/download/, for example:

```
wget https :// download.swift . org/ swift−5.7−release /
      ubuntu2204−aarch64/swift−5.7−RELEASE/swift−5.7−
      RELEASE−ubuntu22.04−aarch64.tar.gz
```

2. Uncompress the downloaded file.

```
tar zxvf ./ swift−5.7−RELEASE−ubuntu22.04−aarch64.tar.gz
```

3. Export the path of Swift to an environment variable.

```
export PATH=~/swift−5.7−RELEASE−ubuntu22.04−aarch64/usr/
    bin:\${PATH}
```

4. Download FuzzJIT from GitHub.

```
git clone https :// github .com/SpaceNaN/fuzzjit
```

5. Compile the FuzzJIT.

```
swift build [−c release ]
```

A.3.2 Basic Test

The user can run the following command to see if FuzzJIT works.

```
swift run FuzzilliCli −−help
```

A.4 Evaluation workflow

We list the operational steps and experiments to evaluate FuzzJIT.

A.4.1 Major Claims

Our paper makes four major claims.

C1: FuzzJIT can be used to uncover new bugs in the JavaScriptCore/V8/Spidermonkey/ChakraCore. This is proven by the experiments (E1).

C2: FuzzJIT can achieve better code coverage growth than baselines. This is proven by the experiments (E2).

C3: FuzzJIT can achieve better syntax correctness rate than baselines. This is proven by the experiments (E3).

C4: FuzzJIT can achieve relatively good throughput than baselines. This is proven by the experiments (E4).

A.4.2 Experiments

E1: Finding bugs in testing subjects. One week of fuzzing should work:
How to: Fuzzing given targets for about one week or longer to see any crashes triggered.
Preparation and execution: To fuzz JavaScriptCore with FuzzJIT:
1. Download JavaScriptCore.

```
git clone https :// github .com/WebKit/webkit
```

2. Apply Targets/JavaScriptCore/Patches/*. This step will be a little bit tricky. When the version does not match, the user needs to manually apply the patch.
3. Run the Targets/JavaScriptCore/fuzzbuild.sh script in the WebKit root directory.
4. FuzzBuild/Debug/bin/jsc will be the JavaScript shell for the fuzzer.
5. Fuzz JavaScriptCore.

```
swift run −c release FuzzilliCli −−profile=jsc −−
    timeout=500 −−storagePath=./jsc/ /path/to/webkit/
    FuzzBuild/Debug/bin/jsc
```

To fuzz V8 with FuzzJIT.
1. First download depot_tools.

```
git clone https :// chromium.googlesource.com/chromium/
    tools/depot_tools . git
```

2. Export depot_tools to an environment variable.

```
export PATH=\$PATH:/path/to/depot_tools
```

3. Configure gclient.

```
mkdir v8
cd v8
gclient config https :// chromium.googlesource.com/v8/
    v8
```

4. Synchronize V8's source code.

```
gclient sync
```

5. Run the Targets/V8/fuzzbuild.sh script in the v8 root directory.
6. out/fuzzbuild/d8 will be the JavaScript shell for the fuzzer.
7. Fuzz V8.

```
swift run −c release FuzzilliCli −−profile=v8 −−
    timeout=500 −−storagePath=./v8/ /path/to/v8/out/
    fuzzbuild/d8
```

To fuzz Spidermonkey with FuzzJIT.
1. Download Spidermonkey source code.

```
git clone https :// github .com/mozilla/gecko−dev
```

2. Apply Targets/Spidermonkey/Patches/*. This step will be a little bit tricky. When the version does not match, the user needs to manually apply the patch.
3. Run the Targets/Spidermonkey/fuzzbuild.sh script in the js/src directory of the Firefox checkout.
4. ./fuzzbuild_OPT.OBJ/dist/bin/js will be the JavaScript shell for the fuzzer.
5. Fuzz Spidermonkey.

```
swift run −c release FuzzilliCli −−profile=
    spidermonkey −−timeout=500 −−storagePath=./
    spidermonkey/ /path/to/gecko−dev/js/src/
    fuzzbuild_OPT.OBJ/dist/bin/js
```

To fuzz ChakraCore with FuzzJIT.
1. Download ChakraCore source code.

```
git clone https :// github .com/chakra−core/ChakraCore
```

2. Apply Targets/ChakraCore/Patches/*. This step will be a little bit tricky. When the version does not match, the user needs to manually apply the patch.
3. Run the Targets/ChakraCore/fuzzbuild.sh script in the ChakraCore directory.
4. FuzzBuild/Debug/ch will be the JavaScript shell for the fuzzer.
5. Fuzz ChakraCore.

```
swift run −c release FuzzilliCli −−profile=chakracore
    −−timeout=500 −−storagePath=./chakracore/ /path/
    to/chakracore/FuzzBuild/Debug/ch
```

Results: Fuzzing is a random procedure, but enough time of fuzzing should reproduce the crashes.

E2: Evaluating code coverage. One week of fuzzing should work:
How to: FuzzJIT/Fuzzilli update the fuzzing status per minute, as shown in Figure 1. We can read the code coverage information from the "Coverage:" row of the FuzzJIT/Fuzzilli interface after one week of fuzzing. For Jsfunfuzz, we fail to obtain its coverage information. For Superion/DIE, which are AFL based, we can also read the coverage information from the "map density" row from their interface.

E3: Evaluating syntax correctness rate. One week of fuzzing should work:
How to: FuzzJIT/Fuzzilli update the fuzzing status per minute. We can read the sample syntax correctness rate information from the "Correctness Rate:" row of FuzzJIT interface after one week of fuzzing. For Jsfunfuzz/Superion/DIE, we provide a Python script to calculate the syntax correctness rate, which is at `/path/to/FuzzJIT/script/calculate_syntax_error.py`.

E4: Evaluating throughput. One week of fuzzing should work:
How to: FuzzJIT/Fuzzilli update the fuzzing status per minute. We can read the throughput information from the "Total Execs:" row of FuzzJIT interface after one week of fuzzing. For Jsfunfuzz, its throughput is determined by its timeout threshold since almost all samples can not be finished in given time. For Superion/DIE, we can read its throughput information from its "total execs:" row of their interface.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220912. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenix%20sec2023/.

```
Fuzzer Statistics
-----------------
Total Samples:              45
Interesting Samples Found:  41
Valid Samples Found:        42
Corpus Size:                42
Correctness Rate:           93.33%
Timeout Rate:               0.00%
Crashes Found:              0
Timeouts Hit:               0
Coverage:                   9.54%
Avg. program size:          54.11
Connected workers:          0
Execs / Second:             2.28
Fuzzer Overhead:            0.82%
Total Execs:                938
```

Figure 1: The interface of FuzzJIT.

References

[1] Samuel Groß. Fuzzil: Coverage guided fuzzing for javascript engines. *Department of Informatics, Karlsruhe Institute of Technology*, 2018.

[2] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642. IEEE, 2020.

[3] Jesse Ruderman. Fuzzing tracemonkey. https://www.squarefree.com/2008/12/23/fuzzing-tracemonkey/.

[4] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.