FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler

Abstract

We present a novel fuzzing technique, FuzzJIT, for exposing JIT compiler bugs in JavaScript engines, based on our insight that JIT compilers shall only speed up the execution but never change the execution result of JavaScript code. FuzzJIT can activate the JIT compiler for every test case and acutely capture any execution discrepancy caused by JIT compilers. The key to success is the design of an input wrapping template, which proactively activates the JIT compiler and makes the generated samples oracle-aware themselves and the oracle is tested during execution spontaneously. We also design a set of mutation strategies to emphasize program elements promising in revealing JIT compiler bugs. FuzzJIT drills to JIT compilers and at the same time retains the high efficiency of fuzzing. We have implemented the design and applied the prototype to find new JIT compiler bugs in four mainstream JavaScript engines. In one month, ten, five, two and 16 new bugs are exposed in JavaScriptCore, V8, SpiderMonkey, and ChakraCore, respectively, with three demonstrated exploitable.

1 Introduction

Due to the intrinsic complexity of executing Turing-complete languages, JavaScript engines become a security weakness of browsers and are revealed as containing a majority of browser vulnerabilities [39]. JavaScript engines are responsible for parsing, interpreting, compiling, and executing JavaScript code, and their basic workflow is shown in Figure 1. The parser and the bytecode generator work in the pipeline to transform the JavaScript code into Abstract Syntax Tree (AST) and then the bytecode. The bytecode can be executed directly by the interpreter or compiled by the JIT compiler. The JIT compiler is an opt-in module that can be activated when certain JavaScript code or function becomes *hot*, i.e., being invoked a sufficient number of times. It comes to work on the fly, compiles the function into assembly code and optimizes it to speed up the execution. Sometimes, the JIT compiler adopts a multi-tier design, where the compilation and optimization

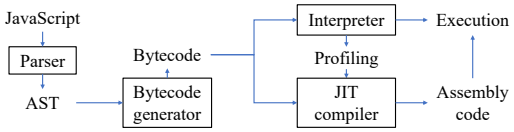

Figure 1: The general workflow of JavaScript engines.

are promoted gradually, as with the increasing number of execution times.

The working mechanism of the JIT compiler, especially the optimization component, is complicated. Hence, there are unsurprisingly many errors in its implementation. Since JavaScript is a weak and dynamically typed language, a direct compilation is not realistic as there are many points the type of a variable is ambiguous. The JIT compiler does not produce a complete compilation of JavaScript code, but sticks to the variable types mostly observed historically according to the runtime profiling information collected by the interpreter. For correctness, the compiled outcome has to be guarded by type checks, and only be used on type compliance. Based on the compilation, the optimizer intends to reduce the number of instructions required to complete functionality. Commonly used optimizations include control flow graph simplification, common subexpression elimination, and dead code elimination [10]. Usually, rigorous control and data flow analyses are required to eliminate unnecessary codes and checks safely. The consequences become serious when some essential security checks are removed incorrectly.

JIT compilers shall only speed up the execution; any changes caused to the execution logic or execution results of JavaScript code manifest JIT compiler bugs. Like logic bugs, many JIT compiler bugs do not lead to a program crash, thus they can be easily missed by fuzzers using crashes as the only oracle, yet these *silent* bugs still provide substantial exploitation primitives. For example, the off-by-one bug [29] in the V8 JIT compiler is exploited to execute code remotely without triggering any crash. More details are discussed in Section 2.3. Besides enhancing the test oracle, to automatically uncover JIT compiler bugs with fuzzing, we must generate test cases that can pass both the syntax and semantic checking, get exe-

cuted, activate the JIT compiler, and intentionally reveal bugs there. As far as we know, the only preliminary attempt to fuzz JIT compiler was from a researcher at Mozilla Security, who extended Jsfunfuzz [28], a generation based fuzzer guided by grammar rules, with a module to examine the printouts of a test case when executed with/without JIT activated in 2008 [27]. It was applied to fuzz Spidermonkey, Mozilla's JavaScript engine, whose JIT activation is designed to be controlled with a pre-defined parameter, and detected 13 JIT bugs at that time. However, it lacks a general JIT compilation triggering mechanism, and the coarse comparison on program final states would make many bugs uncaught. Moreover, the randomly generated test cases are restricted to a limited search space defined by grammar rules and probability and fail to exercise the JIT compiler thoroughly. Recently, some advances have been made on generating syntactically and semantically valid samples, via applying mutations to the AST representation [1, 14, 17, 37], a type-enriched AST representation [12, 23], or a new intermediate representation [9], which supports semantic mutations over control flow and data flow, while keeping the semantic validity. However, none of these advances is specifically designed to test JIT compilers, leaving them thinly activated and tested for a long time. Therefore, a systematically-designed effective fuzzing tool specifically for testing JIT compilers is yet to devise.

In this work, we propose FuzzJIT, a JIT compiler fuzzing technique, which is enhanced by a more precise test oracle that a piece of JavaScript code should produce consistent execution results before and after the JIT compilation, otherwise an error is brought by the JIT compiler. FuzzJIT is featured to proactively activate the JIT compiler, purposefully generate promising inputs that may fail the JIT compiler, and acutely capture those hidden and non-crashing JIT compiler bugs along with crashing bugs. The key to success is the design of an input wrapping template, which makes the generated samples themselves JIT-compiler-activating and oracle-aware, and more importantly, the oracle is tested during execution spontaneously. Besides, we heuristically identify five types of error-prone program elements for the JIT compiler to deal with, and emphasize incorporating them into the generated test cases. As a result, FuzzJIT is able to drill to JIT compilers, unleash its power there, and, at the same time, retain the high efficiency of fuzzing. We evaluate FuzzJIT on four mainstream JavaScript engines, and compare it with four state-of-the-art fuzzers. FuzzJIT stands out in detecting JIT compiler bugs, with ten, five, two and 16 new bugs exposed in JavaScriptCore, V8, SpiderMonkey, and ChakraCore, respectively. It also maintains higher coverage and throughput compared with other baselines.

To summarize, our main contributions include:

- An understanding of common root causes of JIT bugs by studying a large JIT bug corpus.
- An advantageous test case wrapping technique to trigger JIT compilation.
- Test case generation strategies favoring program elements related to the root causes of JIT bugs.
- A novel technique to specifically detect both non-crashing as well as crashing JIT compiler bugs for JavaScript engines.
- A prototype implementation of our approach, FuzzJIT, which is made publicly accessible[1].
- An evaluation on mainstream JavaScript engines, where FuzzJIT exposes 33 new bugs in JIT compilers and shows better performance and bug-finding capability than state-of-the-art fuzzers.

2 Preliminary

2.1 JIT speculative compilation

Traditional compilers trade compilation time for producing assembly code that executes fast at run time. For dynamically typed languages, such as JavaScript, the high-performance compilation technologies cannot be directly applied due to the lack of type information, which guides the compiler to emit assembly code for instructions and allocate registers for inputs and outputs. In the performance war, browsers compete to develop faster JavaScript engines, and *speculative compilation* [25] comes into play to make dynamic languages run faster.

Speculative compilers leverage the insights that, during a particular execution, if a statement is executed with its operands being certain types many times, it is highly likely that it will be executed with the same types more times in the future. Hence, it is worth compiling the statement into the more efficient assembly code conditioned by the type information to speed up the execution. We also say that the assembly code is guarded by the *speculation guard*. Afterward, when the statement is executed again, the JavaScript engine will locate the assembly code and check if the runtime type of the operands matches with the speculation guard, and execute the assembly code on compliance. In case of a mismatch, the engine will roll back to the interpreter or a lower level JIT compiler for execution, also called a bailout. Intuitively, the speculative compilation provides fast passes for frequently seen input types.

Generating compilation for types a statement more frequently executed tends to bring larger improvement on the execution efficiency. To identify these types, when starting executing the JavaScript code, the interpreter is also responsible for collecting the runtime profiling information of variables, e.g., the shape of objects, the type of variables and their values. Once a function or part of its function body is executed sufficient times (according to the threshold set by the JIT compiler), the engine will start the JIT compilation and optimization for this function based on the frequently seen types

[1] The GitHub URL is removed for anonymity reasons.

```
1  function add (a, b){
2      return a + b;
3  }
```

(a) A simple `add` operation of two variables.

```
1  add (a, b):
2      speculateGuard( isInt32 (a));
3      speculateGuard( isInt32 (b));
4      Int32Add(a,b);
5
6  speculateGuard( condition ):
7      if (! condition ): Bailout
```

(b) The execution logic with speculative compilation.

```
1  add (a, b):
2      if ( isInt32 (a)):
3          if ( isInt32 (b)):
4              Int32Add(a,b);
5          else if (isNumber(b)):
6              ConvertToDouble(a);
7              DoubleAdd(a,b);
8          else: Bailout
9      else if (isNumber(a)):
10         if ( isInt32 (b)):
11             ConvertToDouble(b);
12             DoubleAdd(a,b);
13         else if (isNumber(b)):
14             DoubleAdd(a,b);
15         else: Bailout
16     ...
```

(c) The execution logic with interpreter.

Figure 2: The `add` operation and its execution logic.

for type sensitive statements. It is worth noting that the profiled value and type information is also indispensable during the JIT optimization.

Now we demonstrate the speculative compilation with a tiny example. Figure 2a shows a JavaScript `add` operation of two variables, without any type indication. The logic by the JavaScript engine to deal with the `add` operation with and without speculative compilation is illustrated in Figure 2b and Figure 2c, respectively. To ease presentation, we conceptually name a fragment of bytecode implementing certain functionality as an operation, such as `Int32Add` and `isInt32`. Without speculative compilation, the JavaScript interpreter has to consider different scenarios where `a` and `b` are of various runtime types, since the `add` operation is type sensitive. It might be an integer addition, a double addition, a string concatenation, or arbitrary user-defined effects as JavaScript allows rewriting an inherited function. As a result, the interpreter generates expensive control flow logic to handle different cases, as shown in Figure 2c. However, if we observe that the operation is frequently executed with two integers during an execution, a shortcut to execute integer addition can be created. By transforming the code into the speculative compilation as shown in Figure 2b, the number of instructions to execute is greatly reduced. It starts by validating the variable types at runtime against the speculation guard. The `speculateGuard` operation mainly contains a conditional jump to the interpreter or a lower level JIT compiler in case the condition does not hold. If `a` and `b` are indeed integers, a fast pass specialized for integer addition is taken. Otherwise, the JavaScript engine discards the compiled code and bailouts.

2.2 JIT optimization

Based on the speculative compilation, a number of optimizations [25] can be conducted to improve the execution efficiency further. Due to the implementation complexity, these three optimizations – bounds-check elimination, redundancy elimination, and common subexpression elimination – become the principal epicenter of browser vulnerabilities today, both in quantity and quality [10].

Bounds-check elimination. The JavaScript engine applies bounds checks for array indexing operations, during either interpretation or compilation. Bounds-check elimination aims to identify and remove unnecessary ones. The key idea is to perform value range analysis [13] on integer variables indicating indices or array lengths and determine their ranges. If the index is always within the bounds of the array size, the checking can be safely removed to reduce the number of instructions to execute. Bugs occur if the range of the index is underestimated or the range of array size is overestimated. Bounds checks could be wrongly eliminated due to such bugs, and lead to security threats. JIT compiler vulnerabilities under this category include CVE-2015-0817, CVE-2015-2712, CVE-2017-2547, CVE-2017-0234, CVE-2018-0769, and the `String.lastIndexOf` off-by-one bug [29].

Redundancy elimination. Redundancy elimination is to remove duplicate security guards, e.g., type validation, on a particular control flow graph path and only keep the first one. It is safe to do when the side effects of operations between the kept guard and the removed guards are precisely captured, and manifested to be free. In other words, the variables in security guards are never modified by operations in between. Precise modeling of side effects is tricky to achieve, for example, stealthy side effects can be deliberately caused during a function call. Bugs occur when an operation is assumed to be side-effect-free but it is not. CVE-2018-4233 and CVE-2017-11802 are typical vulnerabilities caused by inappropriate redundancy elimination.

Common subexpression elimination. Common subexpression elimination shares a similar spirit as redundancy elimination, but aims to avoid calculating the same expression more than once. It keeps just the first and replaces the rest with a direct copy. Again, this is safe to do only when the operations in between are side-effect-free on expression variables. Note that eliminating an expression also abandons its accompanied security checks, if any. CVE-2020-9802 and CVE-2020-9983 are instances where incorrect common subexpression elimination leads to the removal of essential integer overflow checks and further causes out-of-bound access.

2.3 Security implication

JIT compiler bugs are more exploitable than bugs in the parser and the interpreter. To make successful exploitation, an essential step is to prepare the memory layout with desired contents at proper addresses via memory allocation and deallocation. Afterward, when there are any memory corruption errors, e.g., buffer overflow or use-after-free, the prepared memory contents might be accidentally read or executed by another process, whose execution will be affected or hijacked. When it comes to the exploitation of JavaScript engine bugs, craft-

Table 1: The vulnerabilities exploited in the past three years in Pwn2Own.

Year	Safari	Chrome	Edge	Firefox
2021	CVE-2021-30734[⊗]	CVE-2021-21220[□]	CVE-2021-21220[□]	NA[▽]
2020	CVE-2020-9850[□]	NA[▽]	NA[▽]	NA[▽]
2019	CVE-2019-6216[□] CVE-2019-6217[□]	NA[▽]	Unknown[▲]	CVE-2019-9813[□]

[⊗] The bug is located in an add-on module, WebAssembly [40], of JavaScriptCore.
[▲] Unknown means the demonstration is successful but the details of the bug are unknown to the authors of this paper.
[□] The bug is located in the JIT compiler of the corresponding JavaScript engine.
[▽] NA means the target is not successfully exploited or no one challenges the target.

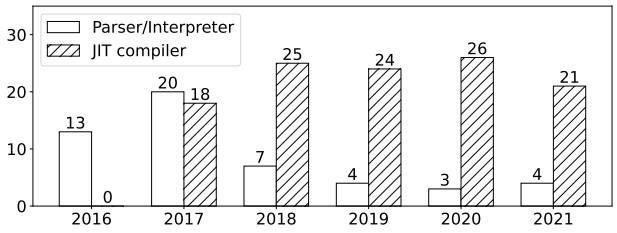

Figure 3: The number of bugs discovered respectively in parser/interpreter and in JIT compiler in recent years.

ing memory layout becomes more practical if the bugs are triggered after the JavaScript code has been put into execution (i.e., at the interpreting or JIT compilation stage). We can conveniently create variable allocation/deallocation statements in the JavaScript code and they get easily executed before reaching the bug point. However, this is not possible for parser bugs. On the other hand, interpreter bugs are also tricky to exploit due to the intensive security checks on the validity of operations, such as boundary and type checking. However, for the sake of execution efficiency, some of these checks will be eliminated by the JIT optimizer, leaving it a security hole and thus more exploitable.

JIT compiler bugs are also popularly exploited to obtain remote code execution in security competitions, such as Pwn2Own[2] and Tianfu Cup[3], and the wild. Table 1 lists all vulnerabilities exploited to control browser targets during the past three years of Pwn2Own. Among the eight successful demonstrations, six of them exploit five vulnerabilities in JIT compilers to obtain remote code execution permission. We also study the JavaScript engine bugs reported by Google Project Zero [8] from 2016 to 2021 (following the identical setups as in [23]), and plot the number of bugs located in the parser/interpreter and the JIT compiler, respectively, in Figure 3. Obviously, more attention and efforts have been put into discovering the JIT compiler bugs in recent years, and the number of them is around four times that of the parser/interpreter bugs during the past four years.

[2]Pwn2Own is a hacking contest held annually at the CanSecWest security conference. Contestants are challenged to exploit widely-used software with previously unknown vulnerabilities.

[3]Tianfu Cup international PWN contest is China's "Pwn2Own", where all teams are required to use original vulnerabilities to hack the given subject.

```
1  function opt() {
2      var maxLen = 268435440; // equals to String::KMaxLength
3      var s = "A".repeat(maxLen);
4      var i = s.lastIndexOf("");
5      // Compiler: i=Range(−1, maxLen−1), Reality: i=Range(−1, maxLen)
6      i += 1;
7      // Compiler: i=Range(0, maxLen), Reality: i=Range(0, maxLen+1)
8      var buf = new Uint8Array(maxLen + 1);
9      return buf[i];
10     // Compiler: Bounds−check removed, Reality: Out−of−bounds access
11 }
12 print(opt()); // undefined
13 %OptimizeFunctionOnNextCall(opt);
14 print(opt()); // out−of−bounds access
```

(a) The PoC causing the out-of-bounds access.

```
1  case kStringIndexOf:
2  case kStringLastIndexOf:
3      return Range(−1.0, String::KMaxLength - 1.0);
```

(b) The implementation root cause in typer.cc.

Figure 4: The String.lastIndexOf off-by-one bug in V8.

3 Motivation

JIT compiler bugs can exist in the speculative compiler or the optimizer. They tend to cause extremely subtle errors at the beginning and require the JIT compiler to be activated and sometimes the fulfillment of pre-conditions on applying the specific defective optimization. They often do not crash the JavaScript engine and thus can be easily missed by fuzzers using crash as the only oracle. However, neglecting these errors will leave JavaScript engines in danger, as they could be exploited and even take control of the engines. In the next, we show how a bounds-check elimination bug in V8 [29] can propagate into a more observable and threatening out-of-bounds access bug.

The PoC causing this bug is shown in Figure 4a. The function `opt` is executed twice, one before the JIT compiler is activated and one after. The line of optimization interest is line 9 where an indexing operation is on the array `buf`. For security reasons, the JavaScript interpreter will check the bounds during interpretation. When the JIT compiler gets activated, it tests whether it is safe to skip the checking by calculating the range of `i` and comparing it with the size of `buf`. The value range analysis follows the data flow of `i` and updates the ranges for each calculation operation modifying it. We highlight the value range of `i` after each statement execution with code comments and report the true range and that calculated by the JIT compiler.

We can see that `i` is initialized at line 4 by calling the `lastIndexOf(toSearch)` of the `String` object. It returns the index of the last occurrence of the string `toSearch`, or `-1` if not found. Here it searches for the empty string over `s`, which is the longest string permitted in JavaScript and filled with the character "A". When searching for the empty string, it will match at the index of `maxLen`, since all strings in JavaScript end up with an empty string by definition. It means,

theoretically, the return value of any call to `lastIndexOf` ranges from `-1` to `String::KMaxLength`. However, the value range analysis in the JIT compiler has mistakenly estimated the upper bound as `String::KMaxLength-1`, as shown in Figure 4b. To exploit this bug, `i` is initialized to be `String::KMaxLength` such that it escapes from the compiler's expectation. After adding `1` at line 6, the estimated range of `i` becomes `[0, maxLen]`, and indexing an array of length `maxLen+1` with `i` is considered always safe. Hence, it becomes optimizable and the compiler removes the bounds-check. However, the actual value of `i` becomes `maxLen+1` after the addition and using it to index an array of `maxLen+1` incurs out-of-bound access.

This bug motivates us from two aspects. First, being more exploitable, JIT compiler bugs are extraordinarily threatening, and detecting them before they get exploited in the wild is of significant importance. Second, JIT compiler bugs can be easily missed by current fuzzing approaches using crash as the only oracle. Just like the `String.lastIndexOf` off-by-one bug, they usually manifest as really subtle errors in the beginning, and require a delicate design to make them propagate and gradually bubble out as observable bugs or even crash. In this example, to make this out-of-bounds access happen, the length of `s` and `buf`, the calling to the `lastIndexOf`, as well as its parameters, and the skew adjustment to `i` at line 6 must be presented exactly this way. Note that even out-of-bounds accesses does not necessarily cause any crash. Hence, a more effective fuzzing approach to detecting JIT compiler bugs, particularly those not triggering crashes, is of demanding need.

4 Approach

To uncover JIT compiler bugs during fuzzing, three challenges have to be overcome when designing the approach: to guarantee that the generated samples evoke the JIT compiler, to enlarge the possibility to reveal bugs there, and, once triggered, to capture the bugs precisely without missing non-crashing bugs and with low false alarms. Overall, we aim to enhance the semantics of generated samples to be more relevant to JIT compilers and the sensitivity of fuzzing tools to JIT compiler errors. In the next, we first overview our approach and then present the mitigation to these three challenges in Section 4.2, Section 4.3, and Section 4.4, respectively.

4.1 Overview

FuzzJIT is the first systematically-designed fuzzing tool to trigger JIT compilers, reveal and capture bugs there, and it is devised to fulfill all above mentioned desired properties. The key to success is a test case template, containing three major components – a JavaScript code snippet for general testing purposes, a JIT compiler trigger, and an oracle-aware verifier based on execution consistency. With this template,

```
1  function deepEquals(r1, r2){ {
2      if(classOf(r1) !== classOf(r2)) return false;
3      ...
4  }
5  function opt(param) {
6      var v0 = [0, 1.0, -1, "a", []];
7      var v1 = new Float32Array(63895);
8      ...
9      if (param){
10         v0 = {x:0x1234, toString:v1};
11         ...
12     }
13     v0[1] = v1;
14     ...
15     return [v0, v1, ...];
16 }
17 var precheck1 = opt(true);
18 for(var i=0; i<5; i++) opt(false);
19 var precheck2 = opt(true);
20 if( deepEquals( precheck1, precheck2 )){
21     var r1 = opt(true);
22     for(var i=0; i<N; i++){   // triggers JIT compiler
23         opt(false);
24     }
25     var r2 = opt(true);
26     if (!deepEquals(r1, r2)){
27         Crash();
28     }
29 }
```

Figure 5: A test generated with the input template of FuzzJIT.

we can automatically enclose any JavaScript test case with the trigger and its verifier and use it to test JIT compilers. A such generated test case is illustrated in Figure 5, where the initial JavaScript test code includes lines 6 to 14 and apparently will not evoke the JIT compiler when fed into the JavaScript engine directly. In the next, we explain how it is wrapped to trigger the JIT compiler and accurately verify the execution consistency based on oracle itself.

To facilitate the multi-time invocations for JIT compiler activation, we wrap the code snippet into a function, named `opt`[4], as shown in line 5. Its argument is specifically designed to test the security guard verification mechanism and will be elaborated on in Section 4.3. Its return value is for detecting non-crashing bugs and will be described later. The function is explicitly called for N times with a `for` loop structure at lines 22 to 24 to trigger JIT compilation. Here N can be customized to any value according to the activation threshold of a particular JIT compiler.

To facilitate the observation of the final state of the function after execution, `opt` returns an array of variables that are modified by the code. Later, a deep comparison is conducted to check whether the final execution states reached before and after the JIT compilation are identical. We report finding a JIT compiler bug on the sighting of a discrepancy. In the template, `opt` is executed without the JIT compiler at line 21, and later it is executed again at line 25 after the JIT compiler gets activated and works to compile (and optimize) the assembly code during lines 22 to 24. More details about the bug capturing design can be found in Section 4.4.

The overall workflow of FuzzJIT is shown in Figure 6.

[4]We consistently call the wrapper function `opt` throughout the paper.

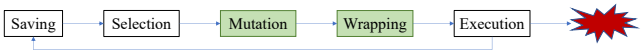

Figure 6: The workflow of FuzzJIT.

Along with the typical fuzzing steps, it features with the code wrapping following the mutation. FuzzJIT also upgrades the mutation module to favor elements of JIT compilation interest, and the details are presented in Section 4.3. Afterward, during the execution, the JIT compiler will be automatically triggered and tested with the enhanced oracle. An alarm emits whenever there is a crash or an execution inconsistency. Last, regardless of whether an alarm is triggered or not, samples triggering new code coverage will be trimmed and saved into the corpus for the next round of fuzzing. The delicate design of wrapping JIT compiler activation and oracle examination to the JavaScript test case makes it a standalone module and can be easily added onto any basic host fuzzer.

4.2 Triggering JIT compiler

The triggering condition of different JavaScript engines differs slightly. Here, we look into four mainstream JavaScript engines, JavaScriptCore, V8, SpiderMonkey, and ChakraCore, which are widely adopted by end users and apply JIT compilation intensively to pursue faster execution speed. Their architectures are shown in Figure 7. Each engine consists of a parser, an interpreter, and one or more JIT compiler tiers. When there are multiple tiers of JIT compilers, they will be activated progressively as the code gets hotter and hotter. Each tier is associated with a specific execution count thresholding the activation; latter tiers tend to have a higher threshold than former tiers and produce assembly code with deeper compilation and optimization. There are a bunch of optimization methods in each tier and they will be optionally activated and interleaved based on the profiling information when the JIT tier gets triggered. Bugs can exist in any tier and a holistic testing approach should be able to drill to each.

The activation thresholds for each JIT compiler tier in different engines are usually configurable. It is worth mentioning that the thresholds cannot be set too small as some optimizations require a least number of execution times to be witnessed when making decisions. Too small thresholds may lead to false positives that cannot be reproduced under the engine's default settings; larger ones sacrifice more testing efficiency. Next, we report the original threshold settings of each engine, and how they are reconfigured, based on trial and error and some industry experience. The configurations work well in our experiments and cause no false positives.

In JavaScriptCore, there are three tiers of JIT compilers. The baseline JIT compiler and the DFG (Data Flow Graph) JIT compiler get activated to compile and optimize a function if it is called more than 6 and 66 times, respectively. The FTL (Faster Than Light) JIT compiler starts to compile if any function runs for more than ten milliseconds on a modern CPU.

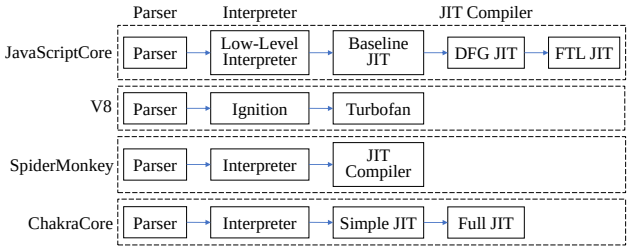

Figure 7: The architecture of four JavaScript engines.

We uniformly customize the thresholds as 10, 50, and 100 invocations to a function in our experiments. In SpiderMonkey, the JIT compiler requires 1,000 times function invocations to trigger, and we set it to 50. In ChakraCore, the Simple JIT and the Full JIT require, respectively, 25 and 20,000 times iterations to trigger by default. We experimentally set them as 10 and 100 times.

When wrapping a test case to trigger the JIT compiler module with a single tier, the template in Figure 5 can be used directly. If there is more than one tier, the triggering structure (similar to lines 21 to 28 in Figure 5) for each tier, from former to latter, will be subsequently pipelined under the true branch of the `if` statement at line 20. To ensure a JIT compiler tier is triggered with a 100% chance during fuzzing, the `opt` function will be called (more than) two times its threshold. That is to say, given a threshold of τ, the N at line 22 in Figure 5 will be set to $2 * \tau$.

The activation mechanism for the V8 Turbofan JIT compiler differs from the other three. Instead of monitoring the number of times a function gets invoked, it tries to predict the benefits of optimizing a function by estimating the amount of time spent executing its unoptimized version and guessing how many times it is to execute in the future [32]. To facilitate testing, V8 provides a builtin native syntax to force the JIT compilation and optimization by calling `%OptimizeFunctionOnNextCall` on the function to compile. Its maintenance team admits bugs found this way regardless of whether they can be reproduced under the original activation setting. Hence, we set N to 1 when fuzzing V8 and explicitly call `%OptimizeFunctionOnNextCall(opt)` right after the `for` loop at line 22.

4.3 Revealing JIT compiler bugs

To make the fuzzing process more effective and expose more bugs, we aim to generate test cases that can substantially challenge the JIT compiler on its correctness of compilation and optimization. Necessary speculation guards are expected to be generated for sensitive operations correctly, and not eliminated inappropriately during optimization. Depending on the error types, targeted input mutation and generation strategies should be developed. Here we discuss bug-leading program elements and structures for fuzzing three of the most popular bug residences, i.e., bounds-check elimination, redun-

dancy elimination, and common subexpression elimination. We propose five heuristic mutation strategies regarding arrays, objects, subexpressions, interesting numbers, and conditioned variable reassignments, as they are the major targets being analyzed during these three types of eliminations.

To confirm the association, especially for arrays, objects, and interesting numbers, we conduct a small empirical study on known JIT bugs and their exploits. We collected 164 distinct JIT compiler bugs along with their PoCs from the Google Project Zero bug report list [8] and a GitHub repository on JavaScript engines CVEs [11], and manually analyzed the existence of these three types of elements. Two authors analyzed all 164 PoCs independently, discussed their discoveries and reached an agreement. Among the 164 bug PoCs, arrays appear in 112 of them, objects appear in 115 of them, and 50 demand special numbers. It is worth mentioning that multiple factors might be required to trigger a bug.

In the next, we elaborate on how the code generation strategies about each element are inspired and designed. Note that the set of strategies could be extended to detect other JIT compiler bugs by analyzing their root causes.

Arrays. Bounds checks at array indexing operations get wrongly eliminated if the index is mistakenly deemed always within the array boundary. This happens when the range of the index or the size of the array is estimated incorrectly during range analysis. The range analysis is more likely to make mistakes on some corner cases, which could be exercised by interesting numbers. The size of an array is affected by APIs conducting operations on it, thus it is important to model and propagate the effect of different APIs on the size range analysis. There are 12 typed arrays in JavaScript, including Int8Array, Uint8Array, Int16Array, etc. For each, there are more than 24 APIs, such as `Array.concat()`, `Array.copy()`, `Array.reverse()`, and some of them need complicated range calculations. For example, when analyzing the array concatenation operation, `Array.concat()`, the range addition should be performed correctly. Among the 164 collected bugs, six including CVE-2014-3176, CVE-2016-1646, and CVE-2017-5030 are primarily caused by the incorrect modeling of `Array.concat()`.

Objects. During redundancy elimination, a type check on a variable will be removed when the variable type is deemed never changed since its last type check. However, modifications of variable types can be conducted in an extremely stealthy way, especially for JavaScript objects, which are the main contributor to type confusion bugs. The type of JavaScript objects is decided by the number and types of their properties. Adding, deleting, or altering the properties of an object will change its type. JavaScript allows all object properties to be altered, even the magical properties, such as `__proto__`, `constructor` and `prototype`. Altering these magical properties allows one to overwrite or pollute the prototype of its base object. If the base object is also inherited by other objects, the pollution will be propagated through the prototype chain. With such stealthy means to modify object types, the type verification algorithm can easily make mistakes.

Subexpressions. The existence of common subexpressions is a must for exercising common subexpression elimination optimization. However, this can hardly or only sparsely be achieved during random mutation. Here, we intentionally make some subexpressions recurring in a test sample, and try to make them complicated by mixing different operations, such as multiplication, division, power, root, and so on. By interspersing these subexpressions at different locations of the program, we challenge the JIT compiler on analyzing the side-effect of operations between any two common subexpressions. If the valuation of the latter subexpression is altered under the table and overlooked by the optimizer, it will be incorrectly replaced with an obsolete value and cause inconsistent execution results.

Interesting numbers. Interesting numbers are incredibly effective in testing corner cases where JIT compilers easily make mistakes especially during range analysis and type check. For example, 268,435,440, the upper bound of the size of a string, is used to trigger the bug in our motivation example [29]. 2.3023e-320 is a special float and it is found to be mistakenly treated as a pointer to an object and trigger type confusion bugs in CVE-2017-11802, CVE-2018-0840, CVE-2018-8556, CVE-2018-0835, CVE-2018-0953, CVE-2018-8466, and CVE-2018-8542. Also, -5.3049894784e-314, which equals to 0x8000000280000002, is the constant `JavascriptNativeFloatArray::MissingItem` and is used to cause type confusion in CVE-2018-0953.

Conditioned variable reassignments. To test whether JIT compilers generate type and bounds checks properly and correctly, never remove them incorrectly during optimization, and verify them correctly at runtime, we design a novel program semantics to alter the type and value of some variables conditionally. We turn off the alteration when triggering the JIT compiler and test whether a proper check can be generated, without seeing the runtime variable type and value at the alteration path during profiling, and survive the optimization if the variable is used in a sensitive operation. This design is illustrated in the template in Figure 5. The `opt` function takes a parameter that controls whether the body of the `if` statement at line 9 gets executed. Inside its body, some variables are changed into different types and values. In this example, `v0` is changed from an array to an object. When triggering the JIT optimization, we carefully control the argument passed to `opt` as `false` (see line 23), such that altered type of `v0` is never observed at runtime. There is an array indexing operation over `v0` at line 13, for which a bounds check and a type check should be generated by the JIT compiler and not removed during optimization, because the side effect of the `if` body can flow to line 13. If the type check is mistakenly removed and `opt` is called with the alteration on (see line 25), the indexing operation over an object will cause a type confu-

sion bug. Otherwise, the type check fails and the JIT compiler should bailout to the lower-level compiler or interpreter for safer processing.

Controlling syntax complexity. When generating the `opt` function body during mutation, we favor the generation of variable declaration and assignment statements, and the JavaScript builtin API call statements associated with various data types. In particular, we increase the chance of generating arrays and their related builtin API calls, and generating objects and their type alteration operations. To make recurring subexpressions, we maintain a pool of existing subexpressions and allow them to be inserted repeatedly during mutation. Also, we extract a set of interesting numbers from the 164 collected PoCs and let the fuzzer choose from it rather than generating random numbers when needed. For the conditioned variable reassignments, we insert an `if` statement conditioned on the parameter of `opt`, and randomly generate its body. To further improve the density of JIT bug revealing elements, we disable the generation of complex statements such as function declarations, class declarations, try/catch statements, switch/case statements. We also avoid generating loop structures inside the function body of `opt`, as it itself is embedded in loops and too many loop structures also hinder the execution efficiency during fuzzing. By keeping the syntax simple and pure, we can also improve the semantic correctness rate of the generated test inputs.

4.4 Capturing JIT compiler bugs

By returning the final execution states of the `opt` function, we can observe its behavior from outside, and ensure that any relevant code shall not be eliminated as dead code during optimization. The testing capability of the code also gets maximized because every statement counts during testing and any tiny error in its execution will fire the alarm. Like typical differential testing [24], we can execute a code fragment and its JIT-ed version, respectively, record their final states and conduct a comparison. This way, their calling context is guaranteed to be identical and any discrepancy is due to the difference between the interpretation and JIT compilation/optimization. However, this will end up with launching the JavaScript engine twice, which is extremely time-consuming and causes terrible lag to the fuzzing process.

Here we propose a novel idea to make a test case self-aware of the execution consistency by integrating these two executions into one single run and including the logic of comparison in itself. The test code (i.e., the `opt` function) is firstly executed purely with the interpreter and then with the JIT compiler on board (see lines 21 to 25 in Figure 5), and their final states are further compared (see lines 26 to 28). This design relies on the important fact that a JavaScript test case is not only a program input to the JavaScript engine, but also some code to be executed. By carefully designing the generation process of the wrapping code, the upgraded test case can be executed successfully, as long as the `opt` function is free of syntax and semantic errors.

Table 2: The comparison rules of `deepEquals()`.

Type	Comparison rule
undefined, null, bigint, symbol, boolean, string	r1 === r2
number	if (r1 === 0) Object.is(r1, r2) else if (isNaN(r1)) isNaN(r2) else r1 === r2
object (Number)	deepEquals(r1.valueOf(), r2.valueOf())
object (Date, String, RegExp, Error, Boolean)	classOf(r1) === classOf(r2) r1.toString() === r2.toString()
object (Array, Map, WeakMap, Set, WeakSet, JSON, Object)	classOf(r1) === classOf(r2) pros1 = Object.keys(r1).sort() pros2 = Object.keys(r2).sort() pros1.length === pros2.length for(var i = 0; i < pros1.length; i++) deepEquals(r1[pros1[i]], r2[pros2[i]])

Now, we explain how to examine if two final states, i.e., the two arrays, are identical. There are eight builtin data types in JavaScript, namely undefined, null, bigint, symbol, boolean, string, number, and object. Except for object, all others are primitive types and we can check whether they are strictly equal with the "===" operator. For a variable of object type, it is a reference/pointer to an object, which is usually associated with a set of properties, in the form of key-value pairs, and a list of methods. Here, keys are of primitive types while values can be of any type, either primitive or non-primitive. To thoroughly compare two objects (see [22]) is costly. Given that the comparison is included in the test case, complicated calculations also slow down the fuzzing speed. Here, we focus on comparing the key-value pairs, as they are the major characters to operate on in a program.

The `deepEquals()` function (see line 1) compares two variables of any type deeply and recursively. Variables of different types are never identical. The inductive rules for comparing variables of the same data types is illustrated in Table 2. Note that the rules are not code and all comparative statement means a comparison check has to be conducted. Two variables are identical if only every comparison encountered during the recursive execution returns true. Rules for primitive types are all base cases and require no recursive call. Particularly, two special cases for the number type are considered, where "===" fails to differentiate `0` and `-0` and deems any comparison involving `NaN` (Not-A-Number) as not equal. Differentiating `0` and `-0` matters in some mathematical calculations, e.g., division and atan2, and several JIT compiler bugs are caused by their misuse, which means it is an informative inconsistency to capture; `Object.is()` is able to tell them apart. `NaN` indicates a failure in mathematical calculation, and its comparison rule with "===" incurs false positives, i.e., unnecessary inconsistency, where two `NaN`s are deemed unequal. Here we rectify it as equal. For object types, we consider commonly used classes, and group them based on whether they could be compared with the same rule. Classes

that mainly wrap a primitive types variable, e.g., `Date` and `String`, can be compared based on their `toString()` values. Here, we convert a `Number` object to its corresponding primitive type precisely with the `valueOf()` function and delegate the comparison to the rules for primitive numbers. Otherwise, two objects are identical if they share the same list of properties (returned by `Object.keys()`), and the value of each property is deeply identical. To balance performance, the `deepEquals()` function considers most of common errors instead of going after capturing all possible inconsistencies.

Eliminate false positives. Executing the same function twice in a single run does necessarily produce the same execution results, even on the same arguments, since their calling context can differ due to some execution side effects, e.g., changing a global variable which is used by the function as well. Other factors that can lead to divergent execution results include generating random numbers (e.g., `Math.random()`), reading current time (e.g., `Date.now()`), and concurrency. Such inconsistency caused should not be counted when auditing the correctness of the JIT compiler. We eliminate these annoying effects to avoid false positives. The concurrency feature can be disabled with command flags for various JavaScript engines. For other factors, we propose a two-pronged approach to settle them efficiently and reliably. We create a blacklist of disturbing APIs and block their generation during mutation. On the other hand, we conduct a pre-check on the existence of these factors by consecutively executing the function a few times and see if there is any discrepancy in the final states (see lines 17 to 20 in Figure 5). Only test cases passing the pre-check will be forwarded to test the JIT compiler. The blacklist improves the success rate of obtaining a valid test case and ensures fuzzing efficiency. To have a complete blacklist is non-trivial, the pre-check is there to effectively cut off unqualified test cases. We also carefully control the execution iterations to be small and avoid activating the JIT compiler.

5 Evaluation

Testing subjects. We select four mainstream JavaScript engines, namely JavaScriptCore (JSC) in Safari, V8 in Chrome, SpiderMonkey (SM) in Firefox, and ChakraCore (CH) in Edge (before March 2021, and currently in maintenance mode), and use their latest build in December 2021 (when we started our experiments) as the testing subjects to evaluate our approach. These JavaScript engines all have large code bases (see Table 6 in Appendix A.1), and have gone through exhaustive manual auditing and testing by their quality assurance teams and wild security researchers. Any new bugs detected by FuzzJIT are escapers from all earlier examinations, which demonstrate the effectiveness of our approach.

Implementation and settings. We implement FuzzJIT based on Fuzzilli [9]. We utilize its essential fuzzing harness, including code coverage feedback and execution result analysis, and customize the input mutation module to generate our JIT bug revealing elements and add the new input wrapping module (see Figure 6). Our evaluation environment is a Ubuntu 20.04 system, running on an i9-10900K CPU with 64GB RAM.

Baselines. We compare FuzzJIT with Jsfunfuzz [28], a JavaScript fuzzer with JIT-specific extension, and three state-of-the-art general-purpose JavaScript engine fuzzers, i.e., Superion [37], DIE [23], and Fuzzilli. Superion, DIE and Fuzzilli are mutation based fuzzing tools. Superion finds bugs by conducting crossover on the AST sub-trees of two parent samples. DIE advances the crossover by restricting the types of sub-tree. Jsfunfuzz generates a large test case (up to 229KB) by embedding almost all JavaScript elements and refreshing some features randomly under each runs. Its JIT-specific extension executes a test case twice, with or without the JIT activated, and identifies a bug if the two printouts are different. Fuzzilli features with fine grained bytecode level mutation to achieve better control and data flow mutation.

Experiment design. We first evaluate the bug-finding capability of FuzzJIT on all testing subjects, and report the bugs detected by FuzzJIT and other baselines, and whether they are bugs in JIT compilers. Then, to better understand the fuzzing effectiveness and efficiency of FuzzJIT, we measure two important metrics, namely code coverage of the generated test cases and the throughput of FuzzJIT and other baselines. Branch coverage reflects the percentage of the branches exercised by the tests samples over the total number of branches; a higher branch coverage implies a more thorough examination. The execution throughput is measured by the number of test cases executed in 24 hours; a larger throughput indicates faster tests execution during fuzzing. High code coverage and throughput are both desired properties of fuzzing tools to hunt more bugs. Second, to better understand the importance of each of the three modules in FuzzJIT – the JIT triggering module, the JIT bug revealing module and the JIT bug capturing module, as well as the fine-grained importance of each mutation strategy in the JIT bug revealing module, to the performance of FuzzJIT, we conduct an ablation study by disabling one and only one of the modules or the mutation strategies and measure the number of bugs discovered and the final code coverage by these variants of FuzzJIT. Last, we case study the bug-triggering tests generated by FuzzJIT and demonstrate the effectiveness of our JIT bug revealing input mutation strategies.

When measuring and comparing the code coverage and throughput, we execute the experiment for multiple campaigns and test the statistical significance of FuzzJIT achieving better performance (i.e., higher coverage/throughput) than the baselines/variants with Vargha Delaney $\hat{A}_{12}$ and Mann Whitney U test (U). Mann Whitney U test (U) tests whether a list of observations is stochastically greater than the other list, while $\hat{A}_{12}$ measures the magnitude of the difference (effect size). The performance difference is significant when the p-value of U (p_U) is below 0.05. $\hat{A}_{12} \geq 0.71$ indicates FuzzJIT outperforms with a large effect size, and $\hat{A}_{12} < 0.29$ means

Table 3: Unique bugs detected during one month by FuzzJIT, baselines, and variants of FuzzJIT.

					Comparison with baselines					Ablation study							
SN	Subject	ID	Output	Module	FuzzJIT	Jsfunfuzz	Superion	DIE	Fuzzilli	-o	-j	-m	-m1	-m2	-m3	-m4	-m5
1	JSC	233XXX*	crash	JIT	✓(1)	-	-	-	-	✓(1)	-	✓(2)	✓(1)	✓(1)	✓(2)	✓(2)	✓(2)
2	JSC	232XXX*	-NaN/NaN	JIT	✓(2)	-	-	-	-	-	-	-	-	-	-	-	✓(2)
3	JSC	232XXX*	1/-1	JIT	✓(4)	-	-	-	-	-	-	-	-	-	-	-	-
4	JSC	228XXX*	True/False	JIT	✓(2)	-	-	-	-	-	-	✓(3)	✓(2)	✓(3)	✓(4)	✓(3)	✓(3)
5	JSC	-	-Infinity/Infinity	JIT	✓(3)	-	-	-	-	-	-	✓(4)	✓(3)	✓(4)	✓(5)	✓(3)	✓(4)
6	JSC	-	255/0	JIT	✓(5)	-	-	-	-	-	-	-	-	✓(5)	-	✓(6)	✓(6)
7	JSC	-	crash	JIT	✓(1)	-	-	-	-	✓(1)	-	✓(2)	✓(2)	✓(3)	✓(1)	✓(2)	✓(4)
8	JSC	233XXX*	undefined/NaN	JIT	✓(1)	-	-	-	-	-	-	-	✓(2)	✓(1)	✓(1)	✓(2)	✓(2)
9	JSC	239XXX*	undefined/NaN	JIT	✓(6)	-	-	-	-	-	-	-	✓(6)	✓(8)	✓(7)	✓(6)	✓(5)
10	JSC	239XXX*	-Infinity/Infinity	JIT	✓(3)	-	-	-	-	-	-	-	✓(3)	✓(5)	✓(4)	✓(3)	✓(5)
11	V8	11XXX$^{\circ}$	True/False	JIT	✓(4)	-	-	-	-	-	-	-	✓(5)	✓(6)	-	✓(5)	✓(4)
12	V8	1224XXX$^{\bullet}$	undefined/123	JIT	✓(3)	-	-	-	-	-	-	✓(3)	✓(3)	✓(4)	✓(6)	✓(4)	✓(3)
13	V8	12XXX$^{\circ}$	14951/14955	JIT	✓(3)	-	-	-	-	-	-	✓(4)	✓(4)	✓(3)	✓(4)	✓(7)	✓(4)
14	V8	1276XXX$^{\bullet}$	crash	JIT	✓(7)	-	-	-	-	✓(7)	-	-	✓(8)	✓(7)	✓(9)	✓(8)	✓(8)
15	V8	12XXX$^{\circ}$	opt()/11	JIT	✓(7)	-	-	-	-	-	-	-	-	-	-	-	-
16	SM	1747XXX$^{\div}$	opt()/NaN	JIT	✓(4)	-	-	-	-	-	-	✓(4)	✓(6)	✓(6)	✓(5)	✓(7)	✓(6)
17	SM	1747XXX$^{\div}$	crash	JIT	✓(6)	-	-	-	-	✓(7)	-	✓(6)	✓(7)	✓(7)	✓(6)	✓(7)	✓(5)
18	CH	6XXX$^{\oplus}$	crash	JIT	✓(1)	-	-	-	-	✓(1)	-	✓(2)	✓(1)	✓(2)	✓(2)	✓(3)	✓(2)
19	CH	6XXX$^{\oplus}$	crash	JIT	✓(2)	-	-	-	-	✓(3)	-	-	-	✓(4)	✓(2)	✓(3)	✓(2)
20	CH	6XXX$^{\oplus}$	crash	JIT	✓(1)	-	-	-	-	✓(2)	-	✓(2)	✓(2)	✓(1)	✓(2)	✓(1)	✓(2)
21	CH	6XXX$^{\oplus}$	crash	JIT	✓(2)	-	-	-	-	✓(3)	-	✓(4)	✓(3)	✓(3)	✓(3)	✓(4)	✓(3)
22	CH	6XXX$^{\oplus}$	crash	JIT	✓(2)	-	-	-	-	✓(3)	-	✓(2)	✓(3)	✓(4)	✓(3)	✓(2)	✓(2)
23	CH	6XXX$^{\oplus}$	crash	JIT	✓(3)	-	-	-	-	✓(5)	-	✓(4)	✓(4)	✓(5)	✓(4)	✓(3)	✓(4)
24	CH	6XXX$^{\oplus}$	crash	JIT	✓(2)	-	-	-	-	✓(3)	-	-	-	✓(3)	✓(5)	✓(5)	✓(3)
25	CH	6XXX$^{\oplus}$	crash	JIT	✓(4)	-	-	-	-	✓(4)	-	✓(5)	✓(4)	✓(5)	✓(5)	✓(4)	✓(4)
26	CH	6XXX$^{\oplus}$	crash	JIT	✓(1)	-	-	-	-	✓(2)	-	✓(2)	✓(1)	✓(2)	✓(2)	✓(2)	✓(3)
27	CH	6XXX$^{\oplus}$	crash	JIT	✓(5)	-	-	-	-	✓(6)	-	-	-	✓(6)	✓(7)	✓(8)	✓(8)
28	CH	6XXX$^{\oplus}$	crash	JIT	✓(2)	-	-	-	-	✓(2)	-	✓(3)	✓(3)	-	✓(3)	✓(2)	✓(4)
29	CH	6XXX$^{\oplus}$	crash	JIT	✓(3)	-	-	-	-	✓(4)	-	✓(4)	✓(3)	✓(5)	✓(6)	✓(3)	✓(4)
30	CH	6XXX$^{\oplus}$	crash	JIT	✓(1)	-	-	-	-	✓(2)	-	✓(2)	✓(3)	✓(1)	✓(2)	✓(3)	✓(1)
31	CH	6XXX$^{\oplus}$	crash	JIT	✓(5)	-	-	-	-	✓(6)	-	-	-	✓(7)	✓(6)	✓(7)	✓(6)
32	CH	059XXX$^{\odot}$	crash	JIT	✓(4)	-	-	-	-	✓(5)	-	✓(7)	✓(7)	✓(6)	✓(6)	✓(7)	✓(4)
33	CH	6XXX$^{\oplus}$	True/False	JIT	✓(2)	-	-	-	-	-	-	-	-	✓(2)	✓(3)	✓(3)	✓(2)
34	CH	6XXX$^{\oplus}$	crash	Interpreter	-	-	✓(3)	-	-	-	-	-	-	-	-	-	-
35	CH	6XXX$^{\oplus}$	crash	Interpreter	-	-	-	✓(5)	-	-	-	-	-	-	-	-	-
36	CH	6XXX$^{\oplus}$	crash	JIT	-	-	-	-	✓(2)	-	-	-	-	-	-	-	-
37	CH	6XXX$^{\oplus}$	crash	Interpreter	-	-	-	-	✓(4)	-	-	-	-	-	-	-	-
38	CH	6XXX$^{\oplus}$	crash	Interpreter	-	-	-	-	✓(5)	-	-	-	-	-	-	-	-
	Total #Bugs (Average #Days to discover a bug)				33(3.09)	0(-)	1(3)	1(5)	3(3.66)	19(3.52)	0(-)	19(3.42)	24(3.58)	29(4.10)	28(4.10)	30(4.16)	31(3.77)

* All bugs have been confirmed on the release versions. Bug ids (last three digits concealed for anonymity reasons) listed above are assigned by Webkit Bugzilla [38](⋆), V8 Bugzilla [33](○), Mozilla Bugzilla [20](÷), and ChakraCore issues list [3](⊕), who are the ordinary maintainers for bugs in JavaScriptCore, V8, SpiderMonkey, and ChakraCore, respectively, and Chromium Bugzilla [6](●) and Microsoft Security Response Center [19](⊙), who provide bug bounty for exploitable bugs.

that FuzzJIT underperforms with a large effect size.

5.1 Comparison experiment

New bugs found. We run all five fuzzers on each testing target for the same period (one month) and compare their ability to find new bugs. The details of the detected bugs are shown in Table 3 (columns 4 to 10). FuzzJIT exposes 33 unique new bugs, among them ten are in JavaScriptCore, five in V8, two in SpiderMonkey, and 16 in ChakraCore. The three bugs for which we created exploitation PoCs have been submitted to Chromium Bugzilla and Microsoft Security Response Center. In comparison, Jsfunfuzz, Fuzzilli, DIE, and Superion find zero, three, one, and one new bug, respectively, and all five bugs are in ChakraCore. Since Fuzzilli, DIE, and Superion are general-purpose fuzzers, they can hardly activate the JIT module, let alone detect bugs there. Disabling the JIT triggering of FuzzJIT also makes it find no bugs, and more details will be discussed in the ablation study. Jsfunfuzz's JIT activating mechanism only works on SpiderMonkey, and due to the limited search space and low throughput (see discussion below), it also failed to spot any new bugs.

For any detected bug, we also report the number of days (in the parentheses in columns 6 to 10) used to find it the first time. On average, FuzzJIT needs only 3.09 days to uncover a bug in the targets. Seven bugs are found on the first day, and eight out of the 33 bugs are exposed by the second day. We can also observe that all bugs detected by FuzzJIT are in the JIT compiler, while among the five bugs detected by other baseline tools, only one detected by Fuzzilli is in the JIT compiler. Here, we check the crash backtrace (tests captured with inconsistency oracle are also forced to crash), and confirm the bug location as the source code closest to the crash. FuzzJIT detects 12 non-crashing bugs and none is false positive. They were accurately captured thanks to the enhanced test oracle and the pre-check mechanism. We also list the values of the particular variable which incurs the discrepancy before and after the JIT compilation in the fourth column.

> Compared with baselines, FuzzJIT exhibits excellent superiority in detecting new JIT bugs in real-world JavaScript engines within the one month fuzzing time window.

Coverage and throughput. For each tool and each testing

Table 4: Branch coverage (10-campaign average) reached after 24-hour fuzzing by FuzzJIT, baselines, and FuzzJIT variants.

Subject	Metric	Comparison with baselines					Ablation study							
		FuzzJIT	Superion	DIE	Jsfunfuzz	Fuzzilli	-o	-j	-m	-m1	-m2	-m3	-m4	-m5
JSC	Average	**21.90%**	16.84%	21.17%	-	16.47%	21.80%	18.09%	19.90%	20.16%	21.11%	20.82%	21.08%	21.65%
	Improvement	-	30.04%	3.48%	-	32.96%	0.45%	21.06%	10.05%	8.63%	3.74%	5.18%	3.88%	1.15%
	$\hat{A}_{12}$	-	0.99	0.81	-	0.99	0.71	0.99	0.99	0.99	0.81	0.99	0.81	0.71
	p_U	-	<0.01	<0.01	-	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
V8	Average	16.67%	12.65%	13.39%	-	13.82%	**16.74%**	15.08%	15.97%	16.41%	16.58%	16.01%	16.21%	16.19%
	Improvement	-	31.77%	24.49%	-	20.62%	-0.41%	10.54%	4.38%	1.58%	0.54%	4.12%	2.83%	2.96%
	$\hat{A}_{12}$	-	0.99	0.99	-	0.99	0.29	0.99	0.99	0.71	0.71	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	-	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
SM	Average	**17.97%**	11.49%	12.41%	-	15.53%	17.89%	14.32%	16.07%	16.36%	16.42%	16.70%	16.17%	16.29%
	Improvement	-	56.39%	44.80%	-	15.71%	0.44%	25.48%	11.82%	9.84%	9.43%	7.60%	11.13%	10.31%
	$\hat{A}_{12}$	-	0.99	0.99	-	0.99	0.71	0.99	0.99	0.99	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	-	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
CH	Average	**22.70%**	15.77%	20.11%	-	19.30%	21.93%	20.41%	19.62%	19.85%	21.65%	19.63%	20.93%	20.72%
	Improvement	-	43.94%	12.87%	-	17.61%	3.61%	11.21%	15.69%	14.35%	4.84%	15.63%	8.45%	9.55%
	$\hat{A}_{12}$	-	0.99	0.99	-	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	-	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01

subject, we measure the branch coverage reached after 24-hour fuzzing, and count the number of tests executed to reflect the throughput. For each setting, 10 campaigns of fuzzing are conducted to mitigate the affect of randomness. We test the statistical significance of our conclusions drawn from comparisons with U and $\hat{A}_{12}$, and report the average value.

The results of branch coverage are presented in Table 4 (columns 3 to 7). The row "Improvement" indicates the percentage of coverage improvement of FuzzJIT compared with a baseline. The detailed instructions used to calculate the branch coverage can be found in Appendix A.3. The branch coverage information for Jsfunfuzz is missing. As a generation-based fuzzing tool without any coverage guidance, Jsfunfuzz does not provide any API to obtain the branch coverage, and we could not find any feasible way to meter its branch coverage, due to its extreme large tests cases. On all subjects, FuzzJIT achieves higher coverage than Superion, DIE and Fuzzilli. As a fuzzer specially for JIT, FuzzJIT also hits superior coverage, as the JIT compilers, concentrated by FuzzJIT, contribute largely to JavaScript engines' codebase, but are thinly explored by other baselines.

> FuzzJIT significantly outperforms Superion, DIE, and Fuzzilli on coverage, with large effect size.

The results of throughput are reported in Table 5. Overall, FuzzJIT executes the most tests in 24 hours on all subjects, while Jsfunfuzz executes the least. Jsfunfuzz generates test cases incorporating massive JavaScript elements, whose size is large and can be up to 100 times of those by FuzzJIT. Superion and DIE demonstrate similar throughput on all four subjects, and Fuzzilli executes a little bit slower than them.

We are a bit surprised to find that FuzzJIT exhibits the best throughput among all tools. FuzzJIT wraps all generated test cases with loop structures to trigger JIT compilers intentionally, which means the tests will execute for a longer time and impede the fuzzing speed. However, FuzzJIT still achieves an exceptionally good throughput. We reckon this is mainly attributed to our code mutation strategies where we avoid generating complex elements and additional loop structures inside the body of `opt`. We also observe that all tools mostly present different throughput on different subjects, which could be due to the intrinsic different implementations of these engines. To confirm the detailed reasons is challenging and out of our research scope.

Table 5: The total number of tests (10-campaign average) executed during 24-hour fuzzing by FuzzJIT and baselines.

Subject	Metric	FuzzJIT	Superion	DIE	Jsfunfuzz	Fuzzilli
JSC	Average	**1,312K**	1,236K	1,245K	8K	1,069K
	Improvement	-	6.14%	5.38%	16300.00%	22.73%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
V8	Average	**1,593K**	1,218K	1,250K	15K	953K
	Improvement	-	30.78%	27.44%	10520.00%	67.15%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
SM	Average	**1,718K**	1,255K	1,224K	12K	1,017K
	Improvement	-	36.89%	40.35%	14216.66%	68.92%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
CH	Average	**3,765K**	3,048K	3,002K	18K	2,932K
	Improvement	-	23.52%	25.41%	20816.66%	28.41%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01

> FuzzJIT achieves the best throughput on all four subject JavaScript engines among all compared tools. Efficient test cases execution also brings a higher chance for FuzzJIT to discover more bugs within the same time budget.

5.2 Ablation study

In this experiment, we create variants of FuzzJIT by disabling one and only one of its three designs, namely, the JIT triggering mechanism(j), the mutation strategies for JIT bug revealing (m), and the enhanced oracle for JIT bug capturing (o), or any one of the five mutation strategies, related to Arrays/Objects (m1), subexpressions (m2), interesting numbers (m3), conditioned variable reassignments (m4), and the control on syntax complexity (m5). We compare the performance of these variants with the vanilla FuzzJIT on two metrics, including the new bugs founded and coverage, and analyze the importance of each design feature and each mutation strategy.

New bugs found. We allocate the same time budget as in the comparison experiment, and execute each variant for one month to detect new bugs. The results are presented in Table 3 (columns 11 to 18). The minus mark in front of a design or mutation indicates that it is disabled. For example, -o means that the oracle enhancement is muted. Overall, disabling any of the designs or mutation strategies, FuzzJIT exposes less bugs and takes longer time to discover them. Also, they do not detect any new bugs that are missed by the vanilla FuzzJIT. Obviously, every design and mutation is crucial to the performance of FuzzJIT. There are two bugs, SN-3 and SN-15, which are not detected by any variant, and we must enable all the designs and mutations to discover them. When disabling the JIT triggering mechanism, *zero* bug is detected on all subjects. Without the enhanced oracle, 14 bugs escape from the detection and only the 19 crash bugs are captured. For the five mutation strategies, their importance to the fuzzing effectiveness varies, where 2 to 9 less bugs are detected, and the detected ones cost longer time to be detected.

All the designs and mutation strategies proposed in this work are essential to success of FuzzJIT.

Coverage. We execute the FuzzJIT variants on all four subjects for 24 hours and repeat for 10 campaigns. The average branch coverage for each setting are reported in Table 4 (columns 8 to 15). Disabling the enhanced oracle makes marginal affect to the coverage. However, a significant reduction on converge is observed on all testing subjects when disabling either the JIT triggering mechanism (-j) or the mutation strategies (-o). In the worst case, the coverage of SpiderMonkey drops by 25.48% without the JIT triggering mechanism, and the coverage of ChakraCore drops by 15.69% without the mutation strategies. Moreover, the removal of any single mutation strategy leads to lower coverage, but the deduction is not as much as removing all of them. The reason is that the probabilities of generating other complex and/or non-interesting elements increase when disabling any mutation strategy, which leads to lower throughput (see Table 8 Appendix B.1), less exercise on the branches, and fewer bugs detected.

Disabling either the JIT triggering mechanism or the mutation strategies, the code coverage drops significantly, while the removal of the enhanced oracle has marginal affect to the coverage.

5.3 Case study

In this section, we case study two bugs detected by FuzzJIT, one in JavaScriptCore and one in bug V8, and both have been fixed in their latest build. The PoCs we present are simplified from the test case generated during fuzzing and are made to fulfill the original JIT compiler activation threshold.

Case study 1. Figure 8 is a simplified PoC triggering Bug-228XXX in JavaScriptCore. The bug is caused by an error

```
1  function opt() {
2    return parseInt("-0");
3  }
4  let r1 = opt();
5  print(Object.is(r1, -0)); // output: True
6  for (let i = 0; i < 1000; i++) {
7    opt();
8  }
9  let r2 = opt();
10 print(Object.is(r2, -0)); // output: False
```

Figure 8: JIT compiler Bug-228XXX in JavaSciptCore.

```
1  function opt() {
2    var v0 = new Int16Array(2**32);
3    return v0[-1];
4  }
5  print(opt()); // output: undefined
6  %OptimizeFunctionOnNextCall(opt);
7  print(opt()); // output: 0
```

Figure 9: JIT compiler Bug-1224XXX in V8.

in the JIT compilation where it generates the bytecode for `parseInt("-0")`. The bug does not cause any crashes and is hard to be detected by other fuzzers. The JavaScript API `parseInt()` converts a string into an integer. Before the JIT compilation, "`-0`" is correctly converted to `-0` at line 4. After the JIT compiler is activated by the repetitive function invocation through line 6-8, `opt` is compiled into bytecode by the JIT compiler and when it is called again, the bytecode converts "-0" to `0`. FuzzJIT captures this inconsistency through the deep comparison rules.

Case study 2. Figure 9 shows a simplified PoC triggering Bug-1224XXX in V8. This bug is also caused by an error during generating bytecode in the JIT compiler, and does not lead to any crash. According to the specification of JavaScript, when reading from an array with an out-of-range index, e.g., `-1`, it should return `undefined`. It is the case at line 5 before the JIT compiler is triggered. After being triggered through line 6, the JIT compiler starts to generate bytecode for the `opt` function. When dealing with the array indexing operation at line 3, the JIT compiler finds that the size of `v0` exceeds the range of the unsigned 32-bit integer type, which is 2^{32}-1. It decides to convert 2^{32} into an unsigned 64-bit integer. The index is required to be of the same type as the array size. Hence, the compiler converts the index `-1` into an unsigned 64-bit integer, without checking its sign, thus making `-1` to be `4,294,967,295` after the conversion, which is 2^{32}-1. Therefore, when `opt` is called again at line 7, the erroneous bytecode is executed, and the value with index 2^{32}-1 in `v0` is returned, which is `0`. Again, FuzzJIT captures this bug with its enhanced test oracle.

Discussion. We find that PoCs triggering these two bugs are quite simple, but they are still missed by other fuzzers. This is mainly due to the fact they use crash as the only test oracle. Also, the interesting number `-0` is crucial to trigger the first bug, which also demonstrates the usefulness of our sample mutation strategy. Due to the limited time budget, we have

not found exploitation of these two bugs ourselves, however it does not mean they cannot be exploited. Though creating exploitation is not within the scope of this research, we did exploit two of our bugs, with ids from Chromium Bugzilla and Microsoft Security Response Center, and do not disclose their details here for security considerations. JIT compiler bugs still have the potential to lead to more threatening security attacks, such as remote code execution, thus should also be paid more attention from the community.

6 Related works

Mutation based approaches. Mutation based approaches produce new test cases from mutating existing test cases at the intermediate representation level [4, 9], the AST level [1, 12, 14, 17, 23, 37], the general grammar level [2, 31], the token level [30], and the byte level [42], which is sorted from JavaScript engine pertinence to generality. Fuzzilli [9] proposes a customized intermediate language named FuzzIL and a set of mutators that can spontaneously ensure the syntax and semantic correctness of generated test cases. PolyGlot [4] proposes a unified intermediate representation for nine programming languages and tests the general fuzzing framework with 21 targets, including JavaScript and its engines. Langfuzz [14] is the first to pursue AST level mutations during fuzzing JavaScript engines. Following this line, Superion [37], Nautilus [1], DIE [23], CodeAlchemist [12], and Montage [17] advance the AST level mutations through introducing code coverage feedback guidance, variable typing system, and prevailing neural network language models. Here, DIE is applied to find crashing JIT bugs during evaluation, but its mutations are not specially designed for revealing JIT bugs. The aim of DIE is to sufficiently unleash the potential of seeds, via largely retaining its structure and variable types. It is highly dependent on the high-quality JIT bug PoC seeds, and can quickly become ineffective if no new PoCs are added. Gramatron [31] performs input mutation based on grammatical automaton and Grimoire [2] uses grammar-like combinations to synthesize new highly structured inputs without any pre-processing step. A recent work on mutating JavaScript tokes [30] is surprisingly effective in finding new bugs in the parser, which are missed by byte-level and grammar-level mutations.

Generation based approaches. Generation based approaches generate test cases based on grammar from scratch. Unlike mutation based approaches, generation based approaches require intensive efforts from experts with thorough domain knowledge and rich experience to develop the generation rules. They tend to generate high-quality samples which could penetrate deeply into the testing targets and sometimes find bugs really hard to be uncovered other way. Jsfunfuzz [28] generates JavaScript test cases from hard-coded grammar rules. It also provides a feature to test the correctness and performance issues of the JIT compiler with differential testing. However, compared to FuzzJIT, it lacks a generic JIT triggering mechanism, mutations specifically-designed to reveal JIT bugs, and a systematically-enhanced test oracle to capture the JIT bugs. And the generation based design also limits its search space. Such inefficiencies make it fail to identify any new bugs in the comparison experiments. Domato [7] generates test cases from given grammar files with associated probabilities. Skyfire [36] learns a probabilistic context-sensitive grammar model from JavaScript corpus to generate new test cases.

Differential testing. Differential testing comparatively tests software systems/applications implementing the same functionality, and is adopted to test JavaScript engines [22, 24, 41] and other software, such as the Java virtual machine [5], SSL/TLS implementations [26, 34], HTTP protocol [16], Java bytecode programs [21], quantum software stacks [35], and CPU [15]. NEZHA [24] is a domain-independent differential testing framework that leverages evolutionary algorithms to guide the generation of samples to maximize the discrepancy between program paths taken during execution. Recent works [22, 41] on JavaScript engine differential testing have focused on exposing conformance bugs, i.e., implementation deviations from the official specification. These approaches are commonly restricted by what oracles can be extracted from the specification, which is still quite challenging. Comfort [41] looks into JavaScript built-ins, extracts expected output from the specification and compares it with the actual execution results. JEST [22] firstly generates the oracle of program final states with a mechanized specification, and embeds the original test input with assertions checking these states, and observes whether there is any assertion failure. The solution of JEST on checking oracle is not as efficient as FuzzJIT, as the same input has to be executed twice to firstly generate the program final state with JIT compiler and instrument the code with assertions accordingly. Efficiency is critical to finding bugs with fuzzing. We can see that none of them is specialized in exposing JIT bugs, and conformance bugs usually have a relatively weak security impact.

7 Conclusion

In this work, we propose an oracle-enhanced fuzzing technique to detect non-crashing and crashing JIT compiler bugs in JavaScript engines. Based on a code wrapping template, we guarantee that all test cases trigger the JIT compiler module and any unexpected execution discrepancy caused by the JIT compiler is captured to identify bugs. Also with the help of a feat of bug-revealing elements, we successfully exposed 33 JIT compiler bugs in four mainstream JavaScript engines. The wrapping template in FuzzJIT is extensible to incorporate other bug-revealing elements and detect JIT compiler bugs of more diverse types.

References

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[2] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1985–2002, 2019.

[3] ChakraCore. Chakracore issues list. https://github.com/chakra-core/ChakraCore/issues/.

[4] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. One engine to fuzz'em all: Generic language processor testing with semantic validation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, 2021.

[5] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of jvm implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–99, 2016.

[6] Chromium. Chromium bugzilla. https://bugs.chromium.org/p/chromium/issues/list.

[7] Ivan Fratric. Domato: A dom fuzzer. https://github.com/googleprojectzero/domato.

[8] Google. Project-zero issues. https://bugs.chromium.org/p/project-zero/issues/list.

[9] Samuel Groß. Fuzzil: Coverage guided fuzzing for javascript engines. *Master's thesis, TU Braunschweig*, 2018.

[10] Samuel Groß. Attacking client-side jit compilers. In *Blackhat USA*, 2018.

[11] Choongwoo Han. A collection of javascript engine cves with pocs. https://github.com/tunz/js-vuln-db.

[12] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[13] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on software engineering*, pages 243–250, 1977.

[14] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 445–458, 2012.

[15] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.

[16] Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin Kirda. T-reqs: Http request smuggling with differential fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1805–1820, 2021.

[17] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language model-guided javascript engine fuzzer. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2613–2630, 2020.

[18] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[19] Microsoft. Microsoft security response center. https://msrc.microsoft.com.

[20] Mozilla. Mozilla bugzilla. https://bugzilla.mozilla.org/.

[21] Yannic Noller, Corina S Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. Hydiff: Hybrid differential software analysis. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1273–1285. IEEE, 2020.

[22] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. Jest: N+ 1-version differential testing of both javascript engines and specification. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 13–24. IEEE, 2021.

[23] 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.

[24] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP)*, pages 615–632. IEEE, 2017.

[25] Filip Pizlo. Speculation in javascriptcore. https://webkit.org/blog/10308/speculation-in-javascriptcore/.

[26] Lili Quan, Qianyu Guo, Hongxu Chen, Xiaofei Xie, Xiaohong Li, Yang Liu, and Jing Hu. Sadt: syntax-aware differential testing of certificate validation in ssl/tls implementations. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 524–535. IEEE, 2020.

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

[28] Jesse Ruderman. Introducing jsfunfuzz. https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/, 2007.

[29] Stephen Röttger. Issue 762874: Security: off by one in turbofan range optimization for string.indexof. https://bugs.chromium.org/p/chromium/issues/detail?id=762874.

[30] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. Token-level fuzzing. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'21)*, 2021.

[31] Prashast Srivastava and Mathias Payer. Gramatron: Effective grammar-aware fuzzing. *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*, 2021.

[32] StackOverflow. After what exact number of execution function gets hot in v8? https://stackoverflow.com/questions/59809832/after-what-exact-number-of-execution-function-gets-hot-in-v8.

[33] V8. V8 bugzilla. https://bugs.chromium.org/p/v8/issues/list.

[34] Andreas Walz and Axel Sikora. Maximizing and leveraging behavioral discrepancies in tls implementations using response-guided differential fuzzing. In *2018 International Carnahan Conference on Security Technology (ICCST)*, pages 1–5. IEEE, 2018.

[35] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. Qdiff: Differential testing of quantum software stacks. In *2021 36nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[36] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[37] 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.

[38] Webkit. Webkit bugzilla. https://bugs.webkit.org.

[39] Wikipedia. Browser security. https://en.wikipedia.org/wiki/Browser_security.

[40] Wikipedia. Webassembly. https://en.wikipedia.org/wiki/WebAssembly.

[41] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. Automated conformance testing for javascript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 435–450, 2021.

[42] Michal Zalewski. American fuzzy lop. https://lcamtuf.coredump.cx/afl/.

A Experiments setting

A.1 Testing subjects

We use four mainstream JavaScript engines, namely JavaScriptCore (JSC) in Safari, V8 in Chrome, SpiderMonkey (SM) in Firefox, and ChakraCore (CH) in Edge (before March 2021, and currently in maintenance mode) as the testing subjects to evaluate our approach. These JavaScript engines all have large code bases, to which the JIT modules make substantial contributions. In Table 6, we list the total number of lines of code and functions in each subject and those in its JIT modules, and we also display the percentage contributed by JIT modules. In terms of lines of code, the occupation of the implementation of JIT modules in the four subjects ranges from 16.67% to 28.91%. On average, the JIT modules account for almost a quarter of all source code lines, which indicates the complexity and importance of JIT compilers. Exploring the JIT compilers and finding bugs there is of significant demand.

Table 6: The size information of the JavaScript engine subjects used in the evaluation.

Subject	Browser	Modules	# Lines	# Functions
JSC	Safari	JIT	57,237	6,357
		Total	255,068	67,861
		JIT ratio	22.43%	9.36%
V8	Chrome	JIT	219,043	21,379
		Total	760,150	116,540
		JIT ratio	28.91%	18.36%
SM	Firefox	JIT	88,054	23,399
		Total	528,049	172,863
		JIT ratio	16.67%	13.53%
CH*	Edge	JIT	61,805	5,375
		Total	246,001	78,995
		JIT ratio	25.12%	6.80%

* ChakraCore was the JavaScript engine for Edge browser before March 2021 and is currently in maintenance mode.

A.2 Configurations for the baseline tools

In our comparison experiments, we use the implementations of baselines provided by UniFuzz [18], which is a fuzzing approach evaluation benchmark and provides a collection of docker for 37 well-known fuzzers, including Superion, DIE, Jsfunfuzz, and Fuzzilli, to ease the evaluation of different fuzzing approaches. The seed and timeout configurations used to execute FuzzJIT and the other four baselines are summarized in Table 7. Only Superion and DIE require initial seeds. DIE's initial seed corpus contains 100 JavaScript files generated by its given script, and the same set of initial seeds are also used for Superion.

Timeout is the max time budget allowed to execute an individual test case. To avoid the fuzzing process getting stuck on non-terminating executions, the timing out mechanism is widely used by fuzzing tools. In principle, a larger timeout is used by fuzzers generating larger inputs. In out experiments, we set the timeout for different tools with the aim to control the timeout rate within 3%. Here, timeout rate is a ratio of the number of executions observed in a period of time that cannot finish within the given timeout over the total number of executions. We meter the execution rate at the end of 24-hour fuzzing, and report the average value of 10 campaigns. For DIE, we keep its default configuration timeout, which is 1000ms. For Superion, Fuzzilli and FuzzJIT, 500ms is set as the timeout limit. The timeout of Jsfunfuzz is set to 50s, which is one or two orders of magnitude larger than others', since its generated samples is super complex and requires a long execution time. Even though, its timeout rate is as high as 32.14%, and we did not set a larger timeout in order to balance the throughput.

Table 7: The seed and timeout configurations of FuzzJIT and other baselines.

Baseline	Jsfunfuzz	Superion	DIE	Fuzzilli	FuzzJIT
#Seed	-	100	100	-	-
Timeout	50s	500ms	1000ms	500ms	500ms
Timeout rate	32.14%	2.45%	1.65%	0.66%	1.37%

A.3 Metering of branch coverage

Different tools offer different ways to get the branch coverage of test samples. For Fuzzilli and FuzzJIT, the branch coverage rate can be easily obtained through their API. For Superion and DIE, the AFL-based fuzzers, the branch coverage cannot be obtained directly. However, it can be calculated from the bitmap information read from the fuzzing statistics. According to AFL technical white paper [42], the branch coverage rate equals to $(Bitmap\ density) * (Bitmap\ size)/(Instrumentation\ count)$, where bitmap size is configured as 2^{22} in Superion and remains default 2^{16} in DIE, the instrumentation count can be calculated by counting the "call __afl_maybe_log" instructions in the instrumented testing subjects, and the bitmap density can be read directly. For Jsfunfuzz, since it is not a coverage-guided fuzzing tool, there is no provided API to extract the branch coverage information. We attempted to calculate it with the help of AFL, however, due to the extreme large tests generated by Jsfunfuzz, AFL crashes due to short of memory (on a machine with 64GB RAM).

B Auxiliary experiment results

B.1 The throughput results in ablation study

In the ablation study, we create variants of FuzzJIT by disabling one and only one of its three designs, namely, the JIT triggering mechanism(j), the mutation strategies for JIT bug revealing (m), and the enhanced oracle for JIT bug capturing (o), or any one of the five mutation strategies, related to Arrays/Objects (m1), subexpressions (m2), interesting numbers (m3), conditioned variable reassignments (m4), and the con-

Table 8: The total number of tests (10-campaign average) executed during 24-hour fuzzing by FuzzJIT and it variants.

Subject	Metric	FuzzJIT	-o	-j	-m	-m1	-m2	-m3	-m4	-m5
JSC	Average	1,312K	1,392K	**1,498K**	1,171K	1,174K	1,226K	1,174K	1,219K	1,166K
	Improvement	-	-5.74%	-12.41%	12.04%	11.75%	7.01%	11.75%	7.62%	12.52%
	$\hat{A}_{12}$	-	0.01	0.01	0.99	0.99	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
V8	Average	2,293K	2,419K	**2,676K**	2,184K	2,188K	2,213K	2,197K	2,191K	2,181K
	Improvement	-	-5.20%	-14.31%	4.99%	4.79%	3.61%	4.36%	4.65%	5.13%
	$\hat{A}_{12}$	-	0.01	0.01	0.99	0.99	0.81	0.81	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
SM	Average	1,718K	1,775K	**1,805K**	1,633K	1,642K	1,643K	1,635K	1,646K	1,626K
	Improvement	-	-3.21%	-4.81%	5.20%	4.62%	4.56%	5.07%	4.37%	5.65%
	$\hat{A}_{12}$	-	0.19	0.01	0.99	0.99	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
CH	Average	3,765K	3,897K	**3,980K**	3,437K	3,522K	3,509K	3,506K	3,544K	3,365K
	Improvement	-	-3.38%	-5.40%	9.54%	6.89%	7.29%	7.38%	6.23%	11.88%
	$\hat{A}_{12}$	-	0.19	0.01	0.99	0.99	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01

trol on syntax complexity (m5). We compare the performance of these variants with the vanilla FuzzJIT on the new bugs founded and coverage. Now, we present how the throughput has been affected.

Each variant is executed for 24 hours on every testing subject, and we count the number of tests executed in total. The process is repeated for 10 times. We report the 10-campaign average in Figure 8 and test the statistical significance and effect size for FuzzJIT outperforming its variants with U and $\hat{A}_{12}$. When the enhanced oracle checking is disabled (-o), there are less instructions to execute and the throughput becomes better. We can observe that the variant without the JIT triggering mechanism (-j) achieves higher throughput than FuzzJIT on all testing subjects. In that case, the tests would not be wrapped into the loop structures, thus becoming more efficient. Our mutation strategies prefer the generation of simple JavaScript elements. When they are muted, the probability of generating them will decline, and in turn the probability of generating other complicated elements will increase. As a result, muting either only one or all of the strategies, the throughput becomes lower than the vanilla FuzzJIT.

Disabling either the JIT triggering mechanism or the enhanced oracle, FuzzJIT hits a higher throughput. While the throughput declines when one or all of the mutation strategies are muted.

B.2 Semantic correctness rate

Generating semantically correct test samples is essential to fuzz the deep area of a program. The semantic correctness rate (SCR) is the percentage of semantically correct tests, i.e., those not raising any uncaught exceptions, among all samples generated during a period of time. A higher SCR is also desirable for uncovering deeply-buried bugs. Here we compare the SCR of FuzzJIT with other baselines. In a similar setting as metering the coverage and throughput in the comparison experiment, we execute each tool on each test subject for 24 hours, and use Fuzzilli to compute the number of all semantically correct samples and total executions. The semantic correctness rate is calculated as the number of semantically correct samples divides the total executions. The process are repeated for ten times, and we further take their average, and report the values in Table 9. We also test the statistical significance and effect size of FuzzJIT achieving a higher SCR than other tools with U and $\hat{A}_{12}$.

Table 9: The semantic correctness rate (10-campaign average) of FuzzJIT and baselines.

Subject	Metric	FuzzJIT	Superion	DIE	Jsfunfuzz	Fuzzilli
JSC	Average	**90.33%**	34.73%	59.21%	45.75%	62.80%
	Improvement	-	160.09%	90.33%	97.44%	43.83%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
V8	Average	**97.04%**	28.54%	57.80%	44.96%	64.34%
	Improvement	-	240.01%	67.88%	115.83%	50.82%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
SM	Average	**93.28%**	47.67%	60.07%	43.55%	64.13%
	Improvement	-	93.29%	55.28%	93.28%	45.45%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
CH	Average	**91.68%**	41.59%	54.04%.	46.71%	62.42%
	Improvement	-	120.43%	69.65%	96.27%	46.87%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
Average (among subjects)		**93.03%**	38.13%	57.78%	45.24%	63.42%

Overall, FuzzJIT achieves the highest SCR on all testing subjects among all the tools, and reaches an average of 93.08%, which is around 46.76% higher than that of the second best tool – Fuzzilli. Fuzzilli demonstrates a good SCR, 63.42% on average. This is mainly credited to its intermediate language which supports semantic mutations. In addition to the merits inheriting from Fuzzilli, FuzzJIT generates test cases following a well-designed structural template, and also carefully controls the mutators to avoid generating complex elements when generating the body of `opt`, thus achieving an even better SCR. Superion shows an average SCR of 38.13%, as its random cross-over mutation operations can easily break the semantic validity. DIE reveals an average SCR of 57.78% on the four targets. Its type-preserving mutation can keep better semantic validity of samples than Superion. Jsfunfuzz

achieves a SCR of 45.24% on average. This low SCR is caused by its generation approach. It first generates a sample, then splits it into two halves and tries to compile each half, mostly aiming to find bugs in the compiler's error-handling mechanisms.

FuzzJIT produces samples with a higher SCR, such that the samples are not turned down before getting interpreted/compiled, drill deeper to JIT compilers, and improve the fuzzing effectiveness.