BCFuzz: Bytecode-Driven Fuzzing for JavaScript Engines

Jiming Wang[*†], Chenggang Wu[*†§], Jikai Ren[*†], Yuhao Hu[*†], Yan Kang[*†], Xiaojie Wei[*], Yuanming Lai[*], Mengyao Xie[*], Zhe Wang[*†]
[*]State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, China
[†]University of Chinese Academy of Sciences
wangjiming21@mails.ucas.ac.cn
{wucg, renjikai22z, huyuhao19f, kangyan, weixiaojie, laiyuanming, xiemengyao, wangzhe12}@ict.ac.cn

***Abstract*—The interpreter and the Just-In-Time (JIT) compiler are two core components of modern JavaScript engines, both of which take bytecodes as input. Most bugs in these components are closely related to specific bytecodes. Therefore, effective fuzzing should pay close attention to how bytecode is generated and exercised. However, previous work fails to consider this aspect and instead focuses primarily on the syntactic and semantic validity of test cases. This causes two major issues: 1) certain bytecodes are never exercised during fuzzing; 2) some bytecodes are exercised infrequently. In this paper, we propose BCFuzz, a bytecode-driven fuzzing approach designed to enhance the diversity of generated bytecode and increase testing opportunities for low-frequency bytecodes. Specifically, we introduce a parser-oriented probing technique to identify the necessary conditions for generating specific bytecodes and use this information to enhance the input generation process. To better test low-frequency bytecodes, we propose bytecode-aware seed preservation, scheduling, and mutation strategies. We evaluate BCFuzz on four mainstream JavaScript engines. In 72 hours of testing, BCFuzz discovers 1.73× and 1.67× more bugs than DIE and Fuzzilli, respectively. In total, BCFuzz uncovered 20 previously unknown bugs. Of these, 17 have already been fixed and one has been assigned a CVE. All the discovered bugs are related to bytecodes.**

***Index Terms*—JavaScript engine, fuzzing, bytecode**

I. Introduction

JavaScript engines are core components of modern browsers, enabling dynamic web interactions [45]. Beyond browsers, they also play a vital role in server-side environments like Node.js [27], desktop applications (e.g., Electron [9]), and mobile apps (e.g., React Native [32]). Vulnerabilities in these engines [31] pose severe security risks across the web and software ecosystems. In browsers, such vulnerabilities can be exploited via malicious web pages to steal sensitive data or, when combined with sandbox escape techniques, achieve remote code execution (RCE) [34]. Therefore, quickly detecting and fixing these vulnerabilities is critical to improving the overall security of JavaScript engines.

The interpreter and JIT compiler are core components of modern JavaScript engines. The interpreter executes JavaScript code by interpreting bytecodes, while the JIT compiler translates frequently executed bytecode into an intermediate representation (JIT IR), performs optimizations, and generates native machine code to improve performance. Both components take bytecodes as input. Although the ECMA-262 standard [8] defines JavaScript's syntax and semantics, it does not standardize bytecode. As a result, each engine adopts its own bytecode format, leading to significant differences across engines, in contrast to Java bytecode, which follows a unified specification.

As JavaScript expands into diverse application domains, vulnerabilities in JavaScript engines have attracted growing attention from both academia and industry. Early fuzzing efforts focused on syntactic validity of test cases, using grammar-based generation (e.g., jsfunfuzz [33], Skyfire [40]) or abstract syntax tree (AST) based mutation (e.g., LangFuzz [18], Superion [41], Montage [20], Nautilus [2]). Later work aimed to enhance semantic validity and diversity. For example, CodeAlchemist [16] introduced definitional constraints, DIE [29] applied typed ASTs, and SoFi [17] leveraged JavaScript reflection mechanism. Fuzzilli [15] and FuzzFlow [44] employed custom intermediate representations to capture program semantics. More recently, differential testing has been widely adopted to identify inconsistencies across engines or between the interpreter and JIT compiler, with tools like JEST N+1 [28], FuzzJIT [42], JIT-Picker [4], and Dumpling [37]. Each of these approaches has uncovered previously unknown bugs.

However, existing fuzzing approaches, whether focusing on the syntactic validity or the semantic validity and diversity of test cases, are all conducted at the JavaScript language level. Generation and mutation at this level provide little control over how bytecode is generated and exercised. Given that both the interpreter and the JIT compiler use bytecode as input, this creates a disconnect between current fuzzing practices and the engine's internal execution model. As a result, during fuzzing: 1) some bytecode may not be exercised at all, and 2) even when exercised, some bytecode may be tested infrequently because existing fuzzing techniques cannot intervene in the generation and execution of specific bytecode. This leads to uneven testing of the engine's bytecode handling logic. We will provide detailed data related to bytecode in Subsection II-B.

The root cause of the above issues is that existing fuzzing approaches overlook bytecode during the testing process. In this paper, we propose BCFuzz, a bytecode-driven fuzzing approach for JavaScript engines that explicitly considers byte-

§ Chenggang Wu is the corresponding author.

code generation and execution. Our method aims to control the bytecode generation process in order to: 1) enhance the diversity of generated bytecode, and 2) increase testing opportunities for bytecode that is infrequently exercised.

Bytecode generation depends on the language constructs present in test cases. To increase bytecode diversity, we first identify the constructs required to trigger specific bytecodes. Since JavaScript engines generate bytecode from the AST produced by the parser, these constructs are embedded in parser behavior. To extract the language constructs associated with different bytecodes, we propose a parser-oriented fuzzing approach. This approach treats the parser of each JavaScript engine as the target, and uses JavaScript tokens as mutation units to probe the parser through testing. By observing parser behavior under varied inputs, we identify the constructs that correspond to the generation of different bytecodes. These constructs are then used to enhance the initial seed corpus or input generation module of existing fuzzers, enabling the generation of test cases that exercise more bytecodes.

To increase the testing frequency of infrequently exercised bytecode, it is crucial to first make the fuzzer aware of both the bytecode generated by test cases and the code related to that bytecode. Therefore, we propose a seed preservation strategy that combines bytecode feedback with edge coverage feedback. A test case is retained as a seed if it generates a previously unseen bytecode. Furthermore, if a test case mutated from this seed can still generate the same bytecode and reach new edges, it is also retained. Additionally, we design a seed scheduling strategy that prioritizes seeds associated with infrequently tested bytecode, and a new mutation strategy. This mutation strategy determines the mutation scope of test cases based on bytecode line number information and performs fine-grained mutations within this scope.

To demonstrate the effectiveness of our approach, we evaluated BCFuzz on the mainstream JavaScript engines, including V8 [14], JavaScriptCore [1], Hermes [10], and JerryScript [19]. BCFuzz discovered 20 new bugs across these four engines, 17 of which have been fixed with one CVE assigned. All identified bugs are related to bytecode, with 90% involving bytecode that was previously untested or rarely tested. Comparative experiments show that, in 72 hours of testing, BCFuzz found $1.73\times$ and $1.67\times$ more bugs than DIE and Fuzzilli, respectively. For the least-tested 5% bytecodes, BCFuzz executed $5.5\times$ more frequently than DIE and $8.9\times$ more frequently than Fuzzilli. Additionally, ablation experiments demonstrate the effectiveness of individual components in BCFuzz. The contributions of this paper are as follows:

- This paper proposes a bytecode-driven fuzzing approach that focuses on the bytecode generation and testing to improve the diversity of generated bytecodes and provide more testing opportunities for infrequently tested bytecodes.
- We found that the parser in all JavaScript engines is responsible for syntactic structure parsing and is closely associated with bytecode generation. To control the generation of specific bytecodes during mutation and generation, we propose a parser-oriented fuzzing approach that automatically obtains the language constructs corresponding to different bytecodes.
- We propose a bytecode-oriented fuzzing strategy for seed preservation, seed scheduling, and mutation that focuses on allocating computational resources to previously untested and under-tested bytecodes.
- Experimental results demonstrate the effectiveness of our approach. BCFuzz successfully discovered 20 bugs across mainstream JavaScript engines.
- We open-source the code to facilitate future research [3].

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

(a) JavaScript add

```
Ldar   a1  // accumulator = a1
Add    a0  // accumulator = accumulator + a0
Return     // return accumulator
```

(b) Bytecode in V8

```
enter
add   dst: loc5, lhs: arg1, rhs: arg2    // loc5 = arg1 + arg2
ret   value: loc5                        // return loc5
```

(c) Bytecode in JavaScriptCore

Fig. 1. `add` bytecode in V8 and JavaScriptCore.

II. Background and Motivation

A. Bytecode in JavaScript Engines

The execution of a JavaScript program in the JavaScript engine is divided into several stages. First, the parser lexically analyzes the JavaScript program to produce a token stream, which is then parsed syntactically into an abstract syntax tree (AST). Next, the bytecode generator converts the AST into bytecodes. The interpreter then takes these bytecodes as input to execute the program. Typically, each bytecode corresponds to a specific interpreter function, for example, `_llint_op_add` function is responsible for interpreting `add` bytecode in JavaScriptCore.

Additionally, the interpreter collects runtime profiling data such as variable types, values and function calls. Once a function or code fragment exceeds a certain execution threshold, the engine invokes the JIT compiler to generate optimized machine code. The JIT compiler takes both bytecode and profiling data as input, translating the bytecode into JIT IR. A single bytecode may yield different JIT IRs depending on operand types. For example, in JavaScriptCore, `add` may translate to `ArithAdd` or `ValueAdd`. The compiler applies various optimization passes on the JIT IR, such as constant folding, dead code elimination and so on. These passes transform the JIT IR while preserving the program's semantics, ultimately producing more efficient machine code.

Unlike Java bytecode, JavaScript bytecode does not follow a standardized format. Each engine defines its own representation. Figure 1 illustrates how the addition operation is encoded in the bytecode of V8 and JavaScriptCore. V8 uses an implicit accumulator for operands, resulting in a more compact format, while JavaScriptCore relies on explicit registers. Moreover, the same language construct may produce different bytecodes across engines. For example, the method call `obj.foo()` is compiled to `call` in JavaScriptCore, but to the specialized `CallProperty0` in V8.

Different bytecodes in JavaScript engines have varying generation conditions. Some bytecodes, such as `mov` in JavaScriptCore, are triggered by multiple common constructs. For example, `mov` can be generated during variable assignments (`x = y`), when storing intermediate expression results, or while preparing arguments in function calls like `add(1, 2)`. In contrast, other bytecodes are associated with more specific conditions. For instance, `put_getter_by_val` bytecode, which defines a getter for an object property, is generated only when all of the following conditions are met: the current scope is within a class or object, the `get` keyword is present, and the getter's name is enclosed in square brackets.

B. Analysis on Bytecode

In recent years, fuzzing techniques for JavaScript engines have advanced considerably. To ensure syntactic validity, methods like LangFuzz [18], Superion [41], Nautilus [2], and Montage [20] perform mutations on ASTs, while SkyFire [40] uses grammar-based generation. However, these approaches offer limited guarantees on semantic validity and diversity. To address this, DIE [29] applies mutations on typed ASTs, preserving structural and type information from seeds. Fuzzilli [15] and FuzzFlow [44] introduce custom intermediate representations, *FuzzIL* and *FlowIR*, and design specialized mutation strategies around them. SoFi [17] further improves semantic diversity by utilizing JavaScript's reflection mechanism in mutation.

These fuzzing approaches have uncovered many bugs. However, current techniques, whether targeting syntactic validity or semantic validity and diversity, operate mainly at the JavaScript language level. Since bytecode is the actual input to the JavaScript engine's interpreter and JIT compiler, existing fuzzing approaches cannot effectively control the generation and testing of bytecode. Although language-level fuzzing can generate bytecode, it often leads to biased distributions. Mutations performed at the JavaScript language level may introduce new execution edges in the parser, yet result in the same bytecode. In such cases, the interpreter and JIT compiler are executing the same logic as before. As a result, some bytecodes are never generated during testing, while some bytecodes are generated less. Consequently, certain parts of the engine's code related to these rarely generated bytecodes receive insufficient testing.

To support this claim, we conducted 72-hour fuzzing experiments with DIE and Fuzzilli on JavaScriptCore. Before testing, we analyzed the engine's code to identify the definition of bytecode and found that JavaScriptCore contains a total of 195 distinct bytecodes. Fuzzilli covered 172, leaving 23 untested; DIE covered 174, leaving 21 untested. Table I lists the 10 most and least frequently tested bytecodes by Fuzzilli, along with the number of test cases that exercised these bytecodes. For statistical convenience, if a test case generates a certain bytecode one or more times, it is counted as a single occurrence of that bytecode being tested. Fuzzilli's top 10 bytecodes averaged 766,325.2 tests, while the bottom 10 averaged only 232.1, highlighting a pronounced imbalance. A similar imbalance exists in DIE.

```
 1  class C1 {
 2    // private field
 3    #aa =42;
 4    f1() {
 5      // bytecode has_private_name
 6      return #aa in this;
 7    }
 8  }
 9  var c = new C1();
10  c.f1();
11
```

(a) Test case 1

```
 1  class C1 {
 2    #aa =42;
 3    f1() {
 4      let v1 = this.#aa;
 5      // let v1 = #aa; //throw exception
 6      // bytecode in_by_val
 7      return v1 in this;
 8    }
 9  }
10  var c = new C1();
11  c.f1();
```

(b) Test case 2

Fig. 2. Language Constructs Required to Generate `has_private_name` Bytecode.

TABLE I
NUMBER OF TIMES BYTECODE HAS BEEN TESTED IN FUZZILLI.

Top Tested Bytecode	Test	Least Tested Bytecode	Test
resolve_scope	784982	new_async_generator_func_exp	880
mov	784982	enumerator_in_by_val	485
get_from_scope	784982	beloweq	210
enter	784982	neq_null	164
end	784982	eq_null	149
call	784982	switch_string	118
put_to_scope	774663	jeq_null	109
construct	741234	nop	100
get_by_id	729437	jneq_null	71
ret	708026	below	35
Average	766325.2	Average	232.1

Through our analysis of JavaScript engines and existing fuzzers, we identified that some bytecodes remain untested because current fuzzers struggle to generate or mutate specific language constructs. For instance, `has_private_name` bytecode in JavaScriptCore is triggered by checking if an object contains a bare private name, as shown in Figure 2(a). This requires the left-hand side of the `in` expression to be a bare private name (e.g., `#aa`), rather than an expression like `this.#aa` or a variable. Fuzzilli uses *FuzzIL* to represent JavaScript programs, where the left-hand side of `in` must be a variable (e.g., `v1`). This results in code like Figure 2(b), which cannot generate the target bytecode. Moreover, bare private names are only valid within `in` expressions. If Fuzzilli inserts such a name outside this context, as in line 5 of Figure 2(b), JavaScriptCore throws an exception. DIE faces a different limitation: its mutation is based on an initial corpus (DIE corpus [6]) that does not contain any language constructs involving private fields, such as `#aa`, making it difficult to construct expressions like `#prop in object`.

Some bytecodes are infrequently tested because their generation depends on strict conditions that current fuzzing techniques struggle to meet. Only specific language constructs can trigger them, and these constructs are easily broken by mutation. For instance, the `below` bytecode in Table I is tested infrequently. In JavaScriptCore, both `below` and `less` represent *less-than* comparisons, but `below` has more restrictive requirements: one operand must be an unsigned right-shift expression, and the other must be either an *Int32* constant or another unsigned right-shift expression. If these conditions are not met, the engine defaults to generating `less`.

Some bytecodes that remain untested or infrequently tested

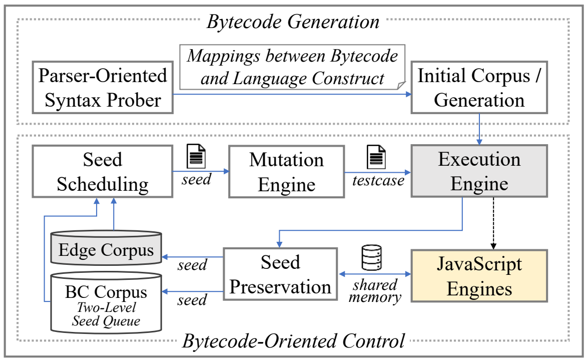

Fig. 3. Workflow of the Bytecode-Driven Fuzzing Approach.

correspond to significant portions of JavaScript engine code. Each bytecode maps to a specific interpreter function in the interpreter. During JIT compilation, each bytecode is translated into JIT IR and participates in various optimizations. To demonstrate this, we manually analyzed JavaScriptCore's interpreter and JIT compiler code for several bytecodes listed in Table I, including those not exercised by Fuzzilli. We recorded the lines of code (LOC) in the relevant interpreter functions and the number of JIT optimization passes applied to the corresponding JIT IRs. Table II presents the results, based on optimization passes from JavaScriptCore's DFG and FTL JIT compilers [30]. The data show that even untested or infrequently tested bytecodes are actively involved in both the interpreter and JIT compiler.

TABLE II
NUMBER OF INTERPRETER FUNCTION CODE LINES AND OPTIMIZATION PASSES RELATED TO EACH BYTECODE.

Category	Bytecode	LoC	Optimizations
Top Tested Bytecode	resolve_scope	127	15
	mov	5	15
	get_from_scope	110	14
	get_by_id	26	15
	ret	6	5
Least Tested Bytecode	new_async_generator_func_exp	10	14
	beloweq	13	11
	neq_null	24	13
	jeq_null	28	14
	below	13	10
Untested Bytecode	has_private_name	17	10
	has_private_brand	12	10
	put_getter_by_val	20	1
	jbelow	16	14
	switch_char	54	8

In the JIT compiler, a single bytecode can produce different JIT IRs based on profiling data. JIT IR provides a more fine-grained representation of the bytecode. Therefore, infrequently tested bytecodes result in insufficient exercise of their corresponding JIT IRs during compilation. In the interpreter, we focus on the generation and testing of bytecode, whereas in the JIT compiler, we focus on the generation and testing of JIT IR. This paper proposes a fuzzing approach that targets bytecodes generation while also considering JIT IRs.

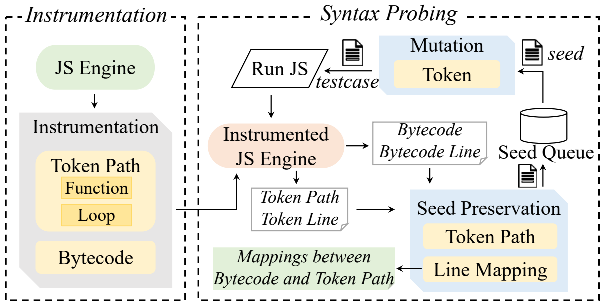

Fig. 4. Workflow of Parser-Oriented Syntax Prober

III. DESIGN

This paper proposes BCFuzz, a bytecode-driven fuzzing approach. BCFuzz aims to improve bytecode (JIT IR) diversity and test infrequently executed bytecodes and JIT IRs. For convenience, throughout Section III and Section IV, the term 'bytecode' is used to denote both bytecode and JIT IR. To ensure the diversity of generated bytecode, the fuzzer must have the capability to control bytecode generation. To enhance the chances of exercising low-frequency bytecodes, the fuzzer should not only focus on generating diverse bytecode but also on executing codes associated with these bytecodes during testing. To this end, we propose a seed preservation strategy that combines bytecode-specific feedback with traditional edge coverage. Additionally, we design new seed scheduling and mutation strategies tailored to bytecodes. Figure 3 illustrates BCFuzz's workflow, which comprises two components: bytecode generation and bytecode-oriented control.

In bytecode generation, we designed a parser-oriented syntax prober that targets the JavaScript engine's parser to extract mappings between bytecodes and their corresponding language constructs. We leverage these mappings to enhance bytecode diversity in the initial seed corpus as well as in test case generation. Deriving these mappings is a non-trivial task. Our detailed approach is presented in Subsection III-A.

In bytecode-oriented control, the execution engine invokes the JavaScript engine to execute test cases from either the enhanced seed corpus or the augmented generation module. After execution, the seed preservation module determine whether the current test case should be stored in *Edge Corpus* or *Bytecode Corpus* (*BC Corpus*). *Edge Corpus* stores test cases that trigger new code edges, while *BC Corpus* retains test cases associated with specific bytecode. The seed scheduling module then selects seeds from either *Edge Corpus* or *BC Corpus* and forwards them to the mutation engine for further mutation. The mutated test cases are subsequently sent to the execution engine. To increase the testing frequency of low-frequency bytecodes, we introduces three bytecode-aware fuzzing strategies: a seed preservation strategy, a seed scheduling strategy, and a mutation strategy, detailed in Subsection III-B.

A. *Parser-Oriented Syntax Prober*

To establish mappings between bytecodes and language constructs, we analyzed JavaScript engine modules, focusing on the parser. The parser module processes the input token

```
1  TreeExpression parseObjectLiteral(TreeBuilder& context) {
2     … // record location
3     consumeOrFail(OPENBRACE);      // token: {
4     if ( consume(CLOSEBRACE) )     // token: }
5        return createObjectLiteral(location);
6
7     TreeProperty property = parseProperty(context);
8     … // create propertyList
9     while (consume(COMMA)) {       // token: ,
10       if (match(CLOSEBRACE))
11          break;
12       property = parseProperty(context);
13       …
14    }
15    consumeOrFail(CLOSEBRACE);
16    return createObjectLiteral(location, propertyList);
17 }
```

(a) Function parseObjectLiteral in Parser.

```
Token Path 1:
OPENBRACE CLOSEBRACE

Token Path 2:
OPENBRACE exe_parseProperty exe_loop CLOSEBRACE

Token Path 3 (in loop):
COMMA call_parseProperty
```

(b) Token Path Examples.

Fig. 5. Parser Function and Token Path Examples.

stream and constructs AST nodes based on specific token combinations. These AST nodes are then used to generate various bytecodes. Based on this observation, we propose a parser-oriented fuzzing approach, implemented as a tool called the parser syntax prober. This approach treats the parser as the fuzzing target and uses JavaScript tokens as mutation units. By fuzzing the parser, we are able to identify token patterns that lead to the generation of specific bytecodes, thereby uncovering the underlying language constructs.

Figure 4 shows the workflow of our parser-oriented fuzzing approach. We instrument the JavaScript engine's parser to collect feedback, including 1) generated bytecodes, 2) bytecode's corresponding line number in the test case, 3) *token path* (to be introduced later), and 4) token's corresponding line number in the test case. We then fuzz the parser to explore language constructs. The mutation module mutates seeds from the seed queue. Although some constructs are hard to generate, tokens in these constructs come from global token set. Hence, we adopt token-level mutation to mutate test cases. The mutated test cases are executed by the instrumented JavaScript engine. To capture syntactic patterns in the language constructs, seed preservation module uses *token path* feedback and establishes a mapping between bytecodes and token paths based on line number information. Figure 5(a) provides an example to illustrate the concept of *token path*.

Figure 5(a) shows a snippet of JavaScriptCore's parser handling object literals. The parser consumes a token stream, checking each token against conditions. For example, line 3 checks if the token is an open brace; if not, parsing fails. If so, it checks for a close brace, parsing an empty object if present; otherwise, it proceeds to parse properties. The parser uses a loop to handle multiple properties. We define the sequence of tokens consumed during parsing as a *token path*. For instance, if the parser executes the code line 3→line 4→line 5, it results in *token path* 1, as illustrated in Figure 5(b).

When parsing different language constructs, the parser follows distinct execution paths, resulting in distinct *token paths*, which indicate the specific construct being processed. We use *token paths* as feedback for seed preservation: if a test case triggers a previously unseen *token path*, it is saved as a new seed. Subsequent token-level mutations on this seed help discover more *token paths*, enabling faster exploration of language constructs. However, using *token paths* as feedback poses two problems:

- 1) Parser functions in JavaScript engines often contain loops and recursive calls. Different numbers of loop iterations or recursion depths can lead to *token paths* of varying lengths, which in turn causes an explosion in the number of unique *token paths* during fuzzing.
- 2) Bytecode generation in JavaScript engines typically involves multiple stages. The parser analyzes syntactic structures and constructs AST nodes, while the bytecode generator emits the corresponding bytecodes based on the specific types of these nodes. Consequently, it is difficult to directly establish a mapping between *token paths* and the generated bytecodes solely from the parser.

Regarding the first problem, the explosion in the number of *token paths* arises from the parser's recursive or iterative processing of certain syntactic structures. Such recursion typically occurs when the same language construct appears in a nested form. For example, when an object literal contains another object literal property, the parser recursively calls `parseObjectLiteral` within `parseProperty` (line 12 in Figure 5(a)). However, multiple recursive calls do not alter the core function behavior, which remains parsing an object literal. Our focus is on the differences within each call, not the recursion depth. To this end, we use intra-function *token paths* as feedback and separate *token paths* generated within each function call, avoiding redundant distinctions caused by recursion depth. Similarly, the parser employs loops to handle sequences of similar constructs with subtle differences. Our focus lies on the execution differences within each loop iteration, rather than on variations resulting from the number of iterations. Therefore, we extract *token paths* generated in each individual iteration and record it separately.

To reconstruct complete language constructs, we insert placeholders in *token path* to mark function calls and loop iterations. As shown in Figure 5(b), *token path* 2 records from the start of the function and inserts placeholders like `exe_parseProperty` for function calls and `exe_loop` for loops. These placeholders help locate where function calls or loop iterations occur within the *token path*, enabling reconstruction of the complete language construct. *Token path* 3 represents a separate path that begins at the start of a loop and stops after completing a single iteration. Recording also stops if parsing fails.

To address the second problem, we map *token paths* to bytecodes via line numbers in the test case. During lexical analysis, each token records its line number, while during bytecode generation, debug information captures the source line number for each bytecode. A *token path* comprises a sequence of tokens that may span multiple lines, so its line numbers form a set, whereas each bytecode typically maps to a single line. If a bytecode's line number contained within a *token path* 's line number set, we can establish a mapping between them. These mappings enhance the initial corpus and input generation of fuzzers to improve bytecode diversity.

B. Bytecode-Aware Fuzzing Strategies

This section introduces three bytecode-aware fuzzing strategies in bytecode-oriented control: a seed preservation strategy combining bytecode feedback with traditional edge coverage feedback, a seed scheduling strategy based on bytecode testing frequency, and a mutation strategy guided by the line number information associated with bytecodes.

1) Seed Preservation: In the interpreter and JIT compiler of JavaScript engines, handling a specific bytecode often involves multiple branches (edges). To test these edges, we adopt a seed preservation strategy that combines bytecode feedback with edge coverage. When a test case generates a new bytecode `B`, it is preserved in *BC Corpus*. We also track mutated test cases derived from this seed. If a mutated test case still generates `B` and covers a new edge, that edge may either (a) part of the code logic that directly handles bytecode `B`, or (b) unrelated to the direct handling of bytecode `B`. In the latter case, two scenarios may arise: (b1) the edge's code has a data dependency with the code handling bytecode `B`, potentially affecting it indirectly; or (b2) the edge is completely unrelated to bytecode `B`. To effectively cover cases (a) and (b1), we retain such mutated test cases as seeds in the *BC Corpus* for further fuzzing. This approach allows bytecodes to guide testing direction while edge coverage refines the granularity of testing, representing the specific code logic.

In *BC Corpus*, BCFuzz maintains a dedicated seed queue for each bytecode, containing both the initial seed that first triggers the bytecode and subsequent mutated test cases that continue to generate the same bytecode while exercising new edges. Additionally, it tracks the testing frequency of each bytecode to guide seed scheduling.

Prior studies [43] [5] [39] [12] [25] [36] have demonstrated that edge coverage feedback is effective in exploring program code. We incorporate this feedback into bytecode-based seed preservation. By leveraging edge coverage, the fuzzer quickly explores the engine's code and discovers new bytecodes. Moreover, since a bytecode may be exercised by different language constructs, edge coverage continues to provide opportunities to uncover those alternatives, even if the bytecode itself has already been exercised. We maintain an *Edge Corpus* to store seeds that trigger new edges.

2) Seed Scheduling: This paper proposes a seed scheduling strategy that prioritizes seeds associated with bytecodes tested less frequently. We design a two-level seed queue within *BC Corpus*. The queue that a seed first enters is referred to as the primary queue. When the fuzzer selects seeds for mutation, it first identifies a subset of bytecodes with the lowest testing frequencies, and randomly chooses seeds from their corresponding seed queues to populate the secondary queue. The fuzzer then randomly selects seeds from the secondary queue for mutation, and removes them from the secondary queue once selected. When the secondary queue becomes empty, it is repopulated from the primary queue based on updated testing frequencies. Additionally, the seed preservation phase maintains a seed queue based solely on edge coverage. When selecting seeds for mutation, BCFuzz selects seeds from either the edge-based seed queue or the bytecode-based secondary queue with a certain probability.

3) Mutation Strategy: To facilitate mutations centered on a specific bytecode, this section introduces a mutation strategy that leverages bytecode line number information.

In Subsection III-A, we established a mapping between *token paths* and bytecodes using their line number information. The line number information of bytecodes can also be used to determine the mutation scope on a seed. During mutation, we first identify the line number corresponding to the newly generated bytecode in the seed. Then, taking this line as the central statement, we perform a data flow-based slicing to determine all statements related to this line, forming the candidate set of statements to be mutated. Finally, the mutator selects statements from this set with a higher probability for mutation.

IV. Implementation

This chapter presents the implementation of BCFuzz. BCFuzz consists of two components: bytecode generation and bytecode-oriented control. The lines of code for each component are listed in Table III.

TABLE III
THE LINES OF CODE OF EACH COMPONENT.

Module	Component	LoC	Language
Bytecode Generation	Instrumentation Pass	1447	C++
	Seed Preservation	549	C
		64	Python
	Generation	1970	Swift
Bytecode-Oriented Control	Seed Preservation	68	C
		190	Swift
	Seed Scheduling	116	C
		448	Swift
	Mutation	164	Swift

A. Parser-Oriented Fuzzing Implementation

The parser-oriented fuzzing is implemented based on AFL++ [11]. AFL++ is an enhanced version of AFL [26], offering a broader set of features. To enable the probing of syntactic structures within the parser, we extend AFL++ with a seed preservation strategy based on *token paths*, while reusing its original token mutator.

To obtain token path information, we instrument the parser module of the JavaScript engine. The parser uses `next_token` to iteratively process tokens. We instrument `next_token` to record tokens during parsing. We implement a *Token Path Analysis Pass* in LLVM [23] to instrument the parser. The recorded *token path* information is stored in shared memory. We record *token path* at function and loop granularity by instrumenting LLVM's *Entry Block* and *Return Block* of functions [24] and the *Header Block* and *Exiting Block* of loops [21]. Recording begins at the *Entry Block* (or *Header Block*) and ends at the *Return Block* (or *Exiting Block*).

B. Bytecode-Oriented Fuzzing Implementation

We implemented the bytecode-oriented fuzzing strategy designed in Subsection III-B on both Fuzzilli [15] and DIE [29], resulting in two versions of BCFuzz, $BCFuzz_F$ and $BCFuzz_D$ respectively. DIE and Fuzzilli represent two different types of fuzzers. DIE performs mutations on AST and requires an initial corpus before testing. Fuzzilli defines a custom intermediate representation called *FuzzIL*, performing generation and mutation on *FuzzIL*. It then translates *FuzzIL* back to JavaScript code through a JSLifter module. Fuzzilli does not require an initial corpus. Both Fuzzilli and DIE are state-of-the-art fuzzers that have discovered many bugs. We implemented our approach on these two fuzzers to demonstrate the effectiveness and generality.

To implement the bytecode-driven seed preservation strategy, we modified Fuzzilli's `evaluate` and DIE's `save_if_interesting` functions. We added three bitmaps: *BCMap*, *LineMap*, and *CountMap*. *BCMap* tracks whether each bytecode is exercised, while *LineMap* records line number information. *CountMap* counts how many times each bytecode has been tested. To implement the bytecode-based seed scheduling strategy, we introduced a two-level seed queue called *BC Corpus* into both Fuzzilli and DIE. To implement the bytecode-based mutation strategy, we modified Fuzzilli's mutation and JSLifter modules.

V. Evaluation

To validate the effectiveness of our approach, this section evaluates BCFuzz and answers the following five questions:

Q1. Can BCFuzz find bugs in the JavaScript engines? How does BCFuzz perform in terms of bug findings comparing to the state-of-the-art fuzzers? (Subsection V-B)

Q2. What are the results of the parser-oriented exploration of language constructs? (Subsection V-C)

Q3. How effective is BCFuzz in testing bytecodes and JIT IRs? Can it cover bytecodes or JIT IRs that were previously untested? Does it provide more testing opportunities for those that were rarely exercised before? (Subsection V-D)

Q4. What is the impact of individual designs in BCFuzz on bug finding? Do these designs make sense? (Subsection V-E)

Q5. How does BCFuzz perform in terms of code coverage against the state-of-the-art fuzzers? (Subsection V-F)

A. Experiment Setup

Baseline: We implement two versions of BCFuzz based on DIE and Fuzzilli, respectively. For a fair comparison, we conduct experiments against DIE and Fuzzilli as baselines. $BCFuzz_D$ and $BCFuzz_F$ require the syntax prober to be executed prior to their run. This procedure is performed only once per engine and constitutes the preparatory step before fuzzing begins. Both DIE and Fuzzilli require preparatory steps before fuzzing. DIE collects a corpus [6], while Fuzzilli continuously extends the FuzzIL language and its generators. Similarly, our syntax prober enriches test inputs by introducing language constructs not covered by either DIE or Fuzzilli, thereby enabling the generation of new bytecodes. After incorporating these constructs, we compare $BCFuzz_D$, $BCFuzz_F$, Fuzzilli and DIE under the same fuzzing time budget to ensure a fair evaluation.

Test Target: We evaluate BCFuzz on four JavaScript engines: JavaScriptCore [1], used in Apple's Safari; V8 [14], powering Google Chrome; Hermes [10], developed by Facebook; and JerryScript [19], a lightweight engine popular in embedded systems. These mainstream engines have been extensively tested by both the security research community and large-scale fuzzing infrastructures like OSS-Fuzz [13].

Environment: We conducted our evaluation experiments on four servers, with their configurations and corresponding experimental tasks summarized in Table IV.

Differential Testing: To accelerate short-term bug discovery, we apply differential testing to identify non-crash bugs [42] [4] [37]. Similar to [4], we implement DiTing, a differential testing system that detects non-crash bugs by analyzing variable value differences between interpreter and JIT compilers.

TABLE IV
Server Configuration.

Machine	CPU	Threads	Memory	Test Target
A	AMD EPYC 7763 2.45GHz	256	256G	JavaScriptCore, Hermes
B	AMD EPYC 7763 2.45GHz	256	256G	V8, JerryScript
C	Intel Xeon Gold 6148 2.4GHz	160	256G	Ablation Experiments - Seed Preservation
D	Intel Xeon Gold 6132 2.6GHz	56	512G	Ablation Experiments - Mutation - Seed Scheduling

B. Bug Finding Ability

1) Bug List: To evaluate BCFuzz's ability to discover previously unknown bugs (Q1), we conducted a four-month testing campaign on four JavaScript engines, each allocated 20 CPU cores. In total, BCFuzz identified 20 new bugs: 13 in JavaScriptCore, 2 new and 1 duplicate in V8, 3 in Hermes, and 2 in JerryScript, as shown in Table V. 17 of these bugs have been fixed and one of them has been assigned a CVE. All bugs are related to bytecodes or JIT IR, with 90% involving bytecodes or JIT IR that were rarely tested before. These are marked with an asterisk (*) in Table V, where "rarely tested" refers to bytecodes/JIT IR with fewer than 20% of the test counts compared to the most frequently tested ones in DIE or Fuzzilli. These findings demonstrate that BCFuzz is effective in uncovering bugs associated with bytecodes and JIT IR.

2) Comparison with Baselines: To evaluate BCFuzz's advantage over existing fuzzers in bug discovery (Q1), we performed experiments using BCFuzz, DIE, and Fuzzilli on

TABLE V
BUGS FOUND BY OPTFUZZ IN JAVASCRIPTCORE (JSC), V8, HERMES AND JERRYSCRIPT (JERRY).

Engine	Issue	Bytecode/JIT IR	Status
JSC	284245§	call/AtomicsIsLockFree*	Fixed
	284612§	call/ArrayPop*	Fixed
	284614§	call/StringValueOf*	Fixed
	284615§	iterator_open*/-	Fixed
	285178§	call/MapIteratorNext*	Fixed
	285179§	enumerator_next*/-	Fixed
	287791‡	enumerator_next/EnumeratorNextUpdate...*	Fixed
	CVE-2025-31215‡	new_array_with_spread*/NewArrayWith...*	Fixed
	288816§	bitand/ArithBitAnd*	Fixed
	288817§	call/StringValueOf*	Fixed
	289959‡	call/NewRegExpUntyped*	Fixed
	291994§	new_array_with_spread/NewArrayWith...*	Fixed
	291997§	enumerator_next*/-	Fixed
V8	406305557§	GetKeyedProperty/LoadDoubleTypedArray...*	-
	408093255‡	CreateCatchContext*/-	Fixed
	crash‡	CreateUnmappedArgs/InlinedAllocation	Duplicate
Hermes	1640§	ReifyArgumentsLoose/CreateArgumentsInst	Fixed
	1664‡	GetArgumentsLength*/-	Fixed
	Bug ...8592‡	GetNextPName*/-	Fixed
Jerry	5223‡	init_class/-	-
	5230‡	bitwise_not*/-	-

‡ in *Issue* column denotes the bug is a crash bug.
§ in *Issue* column denotes the bug is a non-crash bug.
* in *Bytecode/JIT IR* column denotes the bug is associated with bytecodes/JIT IR that were rarely tested.

four JavaScript engines. The versions of the engines and the fuzzers are shown in Table VI. Each test ran for 72 hours and was repeated five times, with each fuzzer allocated 10 CPU cores. We compared the number of bugs discovered by each fuzzer, as shown in Table VII. For crash-triggering bugs, we distinguished unique bugs using stack traces. For non-crashing bugs, we minimized the PoCs and manually classified them. "Total" refers to the total number of unique bugs discovered across the five runs, and "Average" denotes the average number of bugs per run. We also applied the Mann-Whitney U test, denoted as p_U. A value less than 0.05 indicates statistical significance.

TABLE VI
VERSIONS FOR DIFFERENT JAVASCRIPT ENGINES AND FUZZERS.

Category	Engine / Fuzzer	Commit ID	Time
JavaScript Engine	JavaScriptCore	833f10	2024.8.28
	V8	621031	2024.9.2
	Hermes	a49b89	2025.1.8
	JerryScript	502001	2024.12.18
Fuzzer	Fuzzilli	e366dc	2024.5.23
	DIE	91559f	2022.8.14

BCFuzz demonstrates superior bug-finding ability compared to Fuzzilli and DIE. Within the same time budget, $BCFuzz_F$ discovers 2×, 2.5×, and 1.4× more bugs than Fuzzilli on JavaScriptCore, Hermes, and JerryScript, respectively. Similarly, $BCFuzz_D$ finds 1.5×, 3×, 1.5×, and 2× more bugs than DIE on JSC, V8, Hermes, and JerryScript, respectively. On V8, the improvement is less pronounced. Both $BCFuzz_F$ and Fuzzilli discover the same bug, but $BCFuzz_F$ does so more consistently across all five runs. For $BCFuzz_D$, 2 out of 5 runs finds more bugs than DIE. One notable bug found by $BCFuzz_D$ had existed in V8 for over six months and was independently reported by another research team

TABLE VII
BUG RESULTS.

Engine	Metric	$Fuzzilli$	$BCFuzz_F$	DIE	$BCFuzz_D$
JSC	Total	3	6	4	6
	Average	1.8	3.4	2.2	3.6
	p_U	<0.05	-	<0.05	-
V8	Total	1	1	1	3
	Average	0.6	1.0	0.2	0.8
	p_U	0.07	-	0.10	-
Hermes	Total	2	5	4	6
	Average	1.0	2.6	3.4	5.0
	p_U	<0.05	-	<0.05	-
JerryScript	Total	9	13	2	4
	Average	4.4	6.6	2.0	3.2
	p_U	<0.05	-	<0.05	-

Token Path in parseGetterSetter:
OPENBRACKET exe_parseAssignmentExpression CLOSEBRACKET exe_parseFunctionInfo
Bytecode: put_getter_by_val
Language Construct: const foo = { get [2](){} };

Token Path in parseBinaryExpression:
exe_parseUnaryExpression COALESCE exe_loop
Bytecode: mov, jnundefined_or_null
Language Construct: const foo = null ?? "hello";

Fig. 6. Examples of Token Paths and Corresponding Language Constructs.

around the same time. As a result, our bug was classified as a duplicate (CVE-2025-1920). These findings confirm that BCFuzz enhances existing fuzzers' ability to discover bugs, particularly those caused by rarely tested bytecode or JIT IR.

C. Language Construct Exploration

To answer Q2, we present results of parser-oriented syntax probing on JavaScriptCore using two initial corpora: DIE Corpus [6] and a corpus generated by running Fuzzilli on JavaScriptCore for 21 days until coverage nearly saturated, referred to as Fuzzilli Corpus. We ran parser-oriented syntax prober independently on both corpora for 60 hours, repeating each experiment three times.

Table VIII summarizes the results of the syntax prober, including the number of *token path* and mappings initially present in the corpus, as well as those detected by the prober. A mapping denotes a pair of a bytecode and its corresponding token path. For instance, if a token path yields two bytecodes, it results in two mappings. The prober identified 885 additional token paths and 266 new mappings in DIE corpus, and 2,368 additional token paths and 2,196 new mappings in Fuzzilli corpus. Although one token path may correspond to multiple mappings, the number of new mappings is smaller because some token paths do not produce any bytecode. For instance, if a test case contains syntax errors, the JavaScript engine may fail to generate bytecodes. Nevertheless, the syntax prober demonstrates the ability to detect more mappings. Figure 6 shows examples of identified token paths, their associated language constructs and corresponding bytecodes.

To generate and mutate these language constructs, we enhanced Fuzzilli's generation module by introducing 24 new *FuzzILs*, 2 new operators, and refining 10 existing ones. Details are in our repository [3]. We also added 23 new seeds

TABLE VIII
THE RESULTS OF PARSER-ORIENTED SYNTAX PROBER.

Corpus	# of Initial *Token Paths*	# of Initial Mappings	# of Detected *Token Paths*	# of Detected Mappings
DIE	1669	10757	2554	11023
Fuzzilli	1122	13584	3490	15780

to DIE Corpus that serve as inputs for mutations to produce test cases with specific language constructs.

D. Bytecode Testing Results

To answer Q3, we ran BCFuzz, DIE, and Fuzzilli on JavaScriptCore for 72 hours and each is repeated five times. We modified DIE and Fuzzilli to record detailed statistics on distinct bytecodes and JIT IRs exercised and their testing frequency. Table IX summarizes the results, showing the mean values (”Average”) across trials. The ”# of Bytecodes” reports the total number of distinct bytecodes and JIT IRs generated, while ”Avg. Tests Bottom 5%” indicates how frequently the least tested 5% of bytecodes/JIT IRs were exercised.

TABLE IX
NUMBER OF BYTECODES FOUND BY DIFFERENT FUZZERS AND AVERAGE TEST COUNTS FOR THE BOTTOM 5%, 15% AND 25% BYTECODES.

Fuzzer	Metric	# of Bytecodes	Avg. Tests Bottom 5%	Avg. Tests Bottom 15%	Avg. Tests Bottom 25%
DIE	Average	569.8	27.7	122.6	274.2
	p_U	<0.05	<0.05	<0.05	<0.05
$BCFuzz_D$	Average	595.8	152.7	316.0	475.9
	p_U	-	-	-	-
$Fuzzilli$	Average	582.8	90.5	326.4	635.3
	p_U	<0.05	<0.05	<0.05	<0.05
$BCFuzz_F$	Average	592.6	805.0	1436.0	1976.7
	p_U	-	-	-	-

$BCFuzz_D$ generated 26 more unique bytecodes/JIT IRs than DIE, indicating the effectiveness of the additional seeds added to the initial corpus. Likewise, $BCFuzz_F$ produced 10 more bytecodes/JIT IRs than Fuzzilli, demonstrating the contribution of the newly introduced *FuzzIL* instructions. For the least-tested bytecodes/JIT IRs, BCFuzz executed the bottom 5% $5.5\times$ and $8.9\times$ more frequently than DIE and Fuzzilli, respectively. For the bottom 15%, the corresponding execution frequencies were $2.6\times$ and $3.1\times$. The results demonstrate that BCFuzz provides more testing opportunities for bytecodes and JIT IRs that are tested less frequently.

E. Ablation Experiments

To answer Q4, we conducted ablation experiments to evaluate the impact of each design in BCFuzz, including seed preservation, scheduling, and mutation. We selected JavaScriptCore as the test target. Each experiment was run for 72 hours and repeated five times. Table X presents a comparison of various BCFuzz variants in terms of the number of bugs found, the total number of bytecodes and JIT IRs generated, and their average testing frequency. The seed preservation experiment was performed on one server, while the mutation and scheduling experiments were conducted on another. As $BCFuzz_F$ is the baseline in both settings, the table reports two sets of results for $BCFuzz_F$, with differences due to the different hardware configurations.

1) Seed Preservation: To evaluate effectiveness of the seed preservation strategy that integrates edge coverage with bytecode and JIT IR feedback, we modified the seed preservation mechanism of $BCFuzz_F$ and assessed three variants. The first, $BCFuzz_{Gen}$, uses only edge coverage as feedback while supporting additional language constructs compared to Fuzzilli. The second, $BCFuzz_{noedge}$, relies exclusively on bytecode and JIT IR for seed preservation, although it still stores seeds related to edge coverage in *Edge Corpus*. The third, $BCFuzz_{noJITIR}$, maintains the combined feedback strategy but omits JIT IR from seed preservation.

In terms of bug discovery, BCFuzz identifies more bugs than $BCFuzz_{Gen}$, $BCFuzz_{noJITIR}$, and $BCFuzz_{noedge}$, demonstrating the effectiveness of combining edge coverage with bytecode and JIT IR feedback in seed preservation. Regarding the exploration of bytecodes and JIT IRs, BCFuzz, $BCFuzz_{Gen}$, and $BCFuzz_{noedge}$ yield comparable results. For $BCFuzz_{noJITIR}$, only bytecode feedback is used, excluding JIT IRs from seed preservation, resulting in a significantly lower count overall. Typically, interpreters receive more testing than JIT compilers, so bytecodes are tested more frequently. This explains the higher bytecode testing frequency in $BCFuzz_{noJITIR}$. For the bottom 5% least-tested bytecodes, BCFuzz shows higher testing frequency than $BCFuzz_{Gen}$, which lacks bytecode-based seed preservation and thus cannot prioritize less frequently tested bytecodes. $BCFuzz_{Gen}$ achieves higher testing frequency for the bottom 15% and 25% bytecodes, as it does not prioritize the bottom 5% and therefore tests more of the remaining ones.

The seed queue in *BC Corpus* stores seeds that generate the corresponding new bytecode and mutated test cases that continue to produce the bytecode while also triggering new edges. These new edges may be related or unrelated to the bytecode itself. The following data demonstrate this relationship. The *BC Corpus* contains 632 seed queues, with 50 having more than 20 seeds. Among them, 7 are bytecode-related and 43 are JIT IR-related. We analyze 7 bytecode queues to examine the correlation between newly triggered edges and respective bytecodes. Table XI summarizes the results: the “Bytecode” shows the associated bytecode; “Seed Num” refers to the total number of seeds within the queue; “Positive Seed Num” and “Negative Seed Num” denote seeds that trigger relevant and irrelevant edges, respectively; “Positive Rate” is the ratio of Positive Seed Num to Seed Num.

Experimental results indicate that approximately 80% of the seeds generate new edges that include at least one edge related to their corresponding bytecodes. For these seeds, the bytecode/JIT IR guides the testing, while the edges represent the specific logic related to that bytecode/JIT IR. Mutating such seeds effectively exercises the bytecode/JIT IR related logic, which explains why the combined seed selection strategy leveraging both edge coverage and bytecode/JIT IR feedback leads to discovering more bugs.

2) Mutation: To evaluate the effectiveness of the mutation strategy based on bytecode line number information, we replaced it with a random mutation strategy, introducing

TABLE X
ABLATION EXPERIMENT RESULTS.

Metric	Metric	$BCFuzz_F$	$BCFuzz_{Gen}$	$BCFuzz_{noJITIR}$	$BCFuzz_{noedge}$	$BCFuzz_F$	$BCFuzz_{Mutate}$	$BCFuzz_{Schedule}$
# of Bugs	Total	7	5	4	4	9	4	5
	Average	3.2	1.8	1.4	2.4	4.2	2.4	3.0
	p_U	-	<0.05	<0.05	0.14	-	<0.05	<0.05
# of Bytecodes	Average	589.8	588	177.2	590.4	588.0	589.8	590.8
	p_U	-	0.09	<0.05	0.26	-	0.14	0.06
Avg. Bytecode Tests	Bottom 5%	407.6	125.8	2870.4	449.4	2276.3	2151.8	1068.6
	p_U	-	<0.05	<0.05	0.35	-	0.42	<0.05
	Bottom 15%	751.3	1031.3	6171.2	828.8	3431.1	3045.2	3054.4
	p_U	-	<0.05	<0.05	0.16	-	0.21	0.27
	Bottom 25%	1094.2	3000.6	8917.1	1154.9	5476.0	5211.4	5680.1
	p_U	-	<0.05	<0.05	0.16	-	0.58	0.42

TABLE XI
NEW EDGES IN BYTECODE-BASED SEED QUEUE

Bytecode	Seed Num	Positive Seed Num	Negtive Seed Num	Positive Rate
jneq_null	41	22	19	53.7%
jbelow	30	27	3	90.0%
jbeloweq	27	17	10	63.0%
nop	30	25	5	83.3%
eq_null	39	36	3	92.3%
neq_null	24	21	3	87.5%
check_private_brand	26	24	2	92.3%

TABLE XII
CODE COVERAGE RESULTS.

Fuzzer	Metric	Line Cov	Function Cov	Branch Cov
DIE	Average	53.5%	52.7%	36.1%
	p_U	<0.05	0.06	<0.05
$BCFuzz_D$	Average	**54.1%**	**53.0%**	**36.6%**
	p_U	-	-	-
$Fuzzilli$	Average	**52.3%**	**51.2%**	**36.7%**
	p_U	<0.05	<0.05	<0.05
$BCFuzz_F$	Average	51.9%	50.8%	36.1%
	p_U	-	-	-

a variant named $BCFuzz_{Mutate}$. There was no significant difference between $BCFuzz_F$ and $BCFuzz_{Mutate}$ in terms of the number of discovered bytecodes and their testing frequencies. However, $BCFuzz_F$ detected 2.25 times more bugs than $BCFuzz_{Mutate}$. Some bugs related to bytecode or JIT IR are only triggered under specific contexts. By mutating seeds around the bytecode/JIT IR, $BCFuzz_F$ effectively modifies the execution context of the bytecode, enabling the testing of bytecode in diverse contexts. This targeted mutation strategy makes $BCFuzz_F$ more efficient in uncovering bugs.

3) Seed Scheduling: To evaluate the effectiveness of our scheduling strategy based on bytecode testing frequency, we replaced it with a random scheduling approach, introducing a variant named $BCFuzz_{Schedule}$. $BCFuzz_F$ discovered $1.8\times$ more bugs than $BCFuzz_{Schedule}$. For the bottom 5% of least-tested bytecodes/JIT IRs, $BCFuzz_F$ achieved $2.1\times$ higher testing frequency, showing that prioritizing under-tested bytecodes during fuzzing is beneficial for uncovering bugs. For the bottom 15% and 25%, the difference between the two variants was not significant, as $BCFuzz_{Schedule}$ still applies bytecode-based seed preservation and mutation strategies, which guide the fuzzer toward relevant bytecodes. As the scope of tracked bytecodes broadens, the difference between the two approaches gradually diminishes.

F. Code Coverage

To answer Q5, we compared code coverage of BCFuzz, Fuzzilli, and DIE on JavaScriptCore. Each fuzzer ran for 72 hours with five repeats. Coverage metrics including function coverage, branch coverage, and line coverage were collected using *llvm-cov* [22]. Results are summarized in Table XII.

Compared to Fuzzilli, $BCFuzz_F$ achieves lower coverage, likely due to the overhead of bytecode-based trimming before seed preservation, which limits exploration of new edges and reduces coverage. In contrast, DIE does not perform trimming, so adding bytecode feedback has less negative impact on edge exploration. Additionally, the new seeds added to the initial corpus of $BCFuzz_D$ help execute previously unreachable code, resulting in higher coverage than DIE. Despite this, $BCFuzz_F$ and $BCFuzz_D$ find more bugs than their baselines, demonstrating the effectiveness of the bytecode-driven fuzzing approach.

G. Discussion

The number of bugs discovered by BCFuzz varies across JavaScript engines. BCFuzz detects more bugs in JavaScriptCore partly because JavaScriptCore uses specialized JIT IRs for built-in functions, such as `StringValueOf` for `String.prototype.toString`. These IRs are less exercised and more error-prone. BCFuzz is particularly effective at revealing such issues.

Some bytecodes are used only for debugging, requiring special runtime parameters for testing. While we found bugs involving these bytecodes, vendors clarified they are for internal testing and do not acknowledge the issues. Thus, these bytecodes are beyond the scope of this paper.

This paper implements a bytecode/JIT IR-driven fuzzing approach based on DIE and Fuzzilli. Evaluation results confirm its effectiveness. In future work, we plan to extend this approach to other fuzzers to enhance their bug-finding abilities.

VI. RELATED WORK

JavaScript engines are interpreter-based, so only syntactically valid test cases can effectively exercise their functional code. To improve syntactic validity, jsfunfuzz [33] generates programs based on manually written grammar rules. LangFuzz [18] maintains a pool of code fragments and performs assembly and mutation on ASTs. Skyfire [40] learns syntactic patterns from a seed corpus using a probabilistic model to generate valid programs. Superion [41] uses initial seeds crafted from a curated corpus, while Nautilus [2] derives seeds from context-free grammars. Both tools apply mutation and trimming on the AST to produce new test cases.

Beyond syntactic validity, enhancing the semantic validity and diversity of test cases is also essential. CodeAlchemist [16] constructs test cases by combining code fragments that satisfy predefined composition constraints based on variable definitions and references. DIE [29] mutates typed ASTs, retaining structural and type information to preserve semantic correctness. Montage [20] converts ASTs into sequences of subtrees and trains a neural network language model (NNLM) to generate new subtrees for test case replacement. SoFi [17] utilizes JavaScript's reflection mechanism to increase semantic diversity and uses fine-grained static and dynamic analysis to support type-aware mutations. Fuzzilli [15] introduces the intermediate representation *FuzzIL*, enabling mutations that are both syntactically and semantically valid, and incorporates template-based strategies to trigger JIT compilation. FuzzFlow [44] proposes a graph-based IR, FlowIR, which supports direct mutations on the data and control flow of test cases.

Several prior works target specific components within JavaScript engines. Favocado [7] tests the runtime binding layer by generating semantically valid test cases using a combination of statically extracted semantic information and dynamic context. Token-Level Fuzzing [35] focuses on the parser, applying token-level mutations to uncover bugs in syntactic structure parsing and processing. OptFuzz [38] targets JIT optimizations, introducing the concept of optimization trunk paths and proposing seed preservation and scheduling strategies guided by these paths.

In recent years, several works have used differential testing to improve bug detection in JavaScript engines. JEST N+1 [28] applies cross-engine differential testing to find bugs in language specifications and engine implementations. FuzzJIT [42] uses templates to increase JIT compilation triggers and performs differential testing between interpreted and JIT executions at the JavaScript language level. JIT-picker [4] modifies the engine source code and instruments test cases to detect non-crash bugs by comparing variable values in interpreted and JIT runs. Dumpling [37] avoids test case instrumentation by modifying the engine to dump internal states from both the interpreter and JIT compiler at deoptimization points, detecting bugs through differences in these states.

VII. Conclusion

This paper introduces a bytecode-driven fuzzing approach. To improve the diversity of bytecode generation, we propose a parser-oriented fuzzing technique that identifies the language constructs necessary for triggering specific bytecodes. This knowledge is then used to improve the initial corpus or input generation modules of traditional fuzzers. To more effectively target low-frequency bytecodes, we design bytecode-aware strategies for seed preservation, scheduling, and mutation. Evaluation demonstrates the effectiveness of the approach, uncovering 20 previously unknown bugs across JavaScriptCore, V8, Hermes, and JerryScript, 17 of which have been fixed.

Acknowledgments

We thank the anonymous reviewers for their insightful suggestions and comments. This research was supported by the National Natural Science Foundation of China (NSFC) under Grants 62272442, 61902374, U1736208, the CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118), and the Innovation Funding of ICT, CAS under Grant No.E161040.

References

[1] Apple. Javascriptcore, the built-in javascript engine for webkit. https://trac.webkit.org/wiki/JavaScriptCore, 2023.
[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Network and Distributed Systems Security (NDSS) Symposium*, 2019.
[3] BCFuzz. Github. https://github.com/BCFuzz/BCFuzz, 2025.
[4] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. Jit-picking: Differential fuzzing of javascript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 351–364, 2022.
[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.
[6] DIE corpus. https://github.com/sslab-gatech/DIE-corpus, 2020.
[7] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases. In *Network and Distributed Systems Security (NDSS) Symposium*, 2021.
[8] ECMA. Standard ecma-262. https://www.ecma-international.org/publications/standards/Ecma-262.htm, 2024.
[9] Electron. Build cross-platform desktop apps with javascript, html, and css. https://www.electronjs.org/, 2025.
[10] FaceBook. Hermes, a javascript engine optimized for fast start-up of react native apps. https://github.com/facebook/hermes, 2023.
[11] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.
[12] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.
[13] Google. Continuous fuzzing of open source software. https://opensource.google/projects/oss-fuzz, 2023.
[14] Google. Open source javascript and webassembly engine for chrome and node.js. https://v8.dev/, 2023.
[15] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities. In *Network and Distributed Systems Security (NDSS) Symposium*, 2023.
[16] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *Network and Distributed Systems Security (NDSS) Symposium*, 2019.
[17] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, et al. Sofi: Reflection-augmented fuzzing for javascript engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2229–2242, 2021.
[18] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.
[19] JerryScript. A javascript engine for internet of things. https://jerryscript.net/, 2025.

[20] 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.

[21] LLVM. Llvm loop terminology. https://llvm.org/docs/LoopTerminology.html, 2023.

[22] LLVM. llvm-cov. https://llvm.org/docs/CommandGuide/llvm-cov.html, 2024.

[23] LLVM. https://llvm.org/, 2025.

[24] LLVM. Llvm language reference manual. https://llvm.org/docs/LangRef.html, 2025.

[25] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, 2019.

[26] M.Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2019.

[27] Node.js. Node.js is a free, open-source, cross-platform javascript runtime environment. https://nodejs.org/, 2025.

[28] 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.

[29] 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.

[30] Filip Pizlo. Speculation in javascriptcore. https://webkit.org/blog/10308/speculation-in-javascriptcore/, 2020.

[31] ProjectZero. V8 0-day in-the-wild 2021-2022. https://docs.google.com/spreadsheets/d/1lkNJ0uQwbeC1ZTRrxdtuPLCIl7mlUreoKfSIgajnSyY/view?pli=1&gid=0#gid=0, 2022.

[32] Reactjs. https://react.dev/, 2023.

[33] Jesse Ruderman. Introducing jsfunfuzz. https://bugs.chromium.org/p/project-zero/issues/list, 2007.

[34] Saelo. Safari rce, sandbox escape, and lpe to kernel for macos. https://github.com/saelo/pwn2own2018, 2018.

[35] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. {Token-Level} fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2795–2809, 2021.

[36] Dongdong She, Abhishek Shah, and Suman Jana. Effective seed scheduling for fuzzing with graph centrality analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2194–2211. IEEE, 2022.

[37] Liam Wachter, Julian Gremminger, Christian Wressnegger, Mathias Payer, and Flavio Toffalini. Dumpling: Fine-grained differential javascript engine fuzzing. 2025.

[38] Jiming Wang, Yan Kang, Chenggang Wu, Yuhao Hu, Yue Sun, Jikai Ren, Yuanming Lai, Mengyao Xie, Charles Zhang, Tao Li, et al. Optfuzz: optimization path guided fuzzing for javascript jit compilers. In *Proceedings of the 2024 USENIX Security Symposium (SEC'24)*, 2024.

[39] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15, 2019.

[40] 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.

[41] 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.

[42] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. Fuzzjit: Oracle-enhanced fuzzing for javascript engine jit compiler. In *USENIX Security Symposium. USENIX*, 2023.

[43] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[44] Haoran Xu, Zhiyuan Jiang, Yongjun Wang, Shuhui Fan, Shenglin Xu, Peidai Xie, Shaojing Fu, and Mathias Payer. Fuzzing javascript engines with a graph-based ir. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3734–3748, 2024.

[45] Nicholas C Zakas. Professional javascript for web developers, 2005.