Fuzzing JavaScript Engines with Aspect-preserving Mutation

Soyeon Park Wen Xu Insu Yun Daehee Jang Taesoo Kim
Georgia Institute of Technology
{spark720, wen.xu, insu, daehee87, taesoo} @ gatech.edu

***Abstract*—Fuzzing is a practical, widely-deployed technique to find bugs in complex, real-world programs like JavaScript engines. We observed, however, that existing fuzzing approaches, either generative or mutational, fall short in fully harvesting high-quality input corpora such as known proof of concept (PoC) exploits or unit tests. Existing fuzzers tend to destruct subtle semantics or conditions encoded in the input corpus in order to generate new test cases because this approach helps in discovering new code paths of the program. Nevertheless, for JavaScript-like complex programs, such a conventional design leads to test cases that tackle only shallow parts of the complex codebase and fails to reach deep bugs effectively due to the huge input space.**

In this paper, we advocate a new technique, called an *aspect-preserving* mutation, that stochastically preserves the desirable properties, called *aspects*, that we prefer to be maintained across mutation. We demonstrate the aspect preservation with two mutation strategies, namely, *structure* and *type* preservation, in our fully-fledged JavaScript fuzzer, called DIE. Our evaluation shows that DIE's aspect-preserving mutation is more effective in discovering new bugs (5.7× more unique crashes) and producing valid test cases (2.4× fewer runtime errors) than the state-of-the-art JavaScript fuzzers. DIE newly discovered 48 high-impact bugs in ChakraCore, JavaScriptCore, and V8 (38 fixed with 12 CVEs assigned as of today). The source code of DIE is publicly available as an open-source project.[1]

I. INTRODUCTION

Fuzzing is, arguably, the most preferable approach to test complex, real-world programs like JavaScript engines. While conventional unit testing is effective in validating the expected functional correctness of the implementation, automated fuzzing is exceptional at discovering unintended security bugs. According to Google, fuzzers have uncovered an order of magnitude more bugs than handwritten unit tests developed over a decade [4, 21, 33, 40]. For example, fuzzers have discovered over 5000 new bugs in the heavily-tested Chromium ecosystem [37].

Two classes of fuzzers have been developed to test JavaScript engines, namely, *generative* and *mutational* fuzzers. Generative approaches build a new test case from the ground up following pre-defined rules like a context-free grammar of the JavaScript programming language [17, 39] or reassembling synthesizable code bricks dissected from the input corpus [19]; mutational approaches [2, 44] synthesize a test case from existing seed inputs.

However, we observed that both generative and mutational approaches are not enough to take advantage of high-quality input corpora such as known proof of concept (PoC) exploits or existing unit tests. These input corpora are deliberately designed to deal with particular properties of the program under testing, yet such properties are not retained during the fuzzing process. Existing fuzzers [19, 20, 39, 44] are designed to generate naive test cases based on simple generative rules without leveraging input corpora or fail to maintain subtle semantics or conditions encoded in these input corpora when generating new test cases, as destructing them indeed helps in discovering more diverse code paths of the program. Such a design works well for small programs where the input space is tractable for automatic exploration. When testing JavaScript-like complex programs, however, such a conventional design tends to produce test cases that stress shallow parts of the complex codebase, *e.g.*, a parser or an interpreter but not JIT optimization algorithms.

Aspect. In this paper, *aspect* is used to describe a key feature that guides to discover new bugs from the PoC of existing bugs. This term is a different concept from the one in the aspect oriented programming (AOP). Aspect in AOP represents a specific feature that cross-cuts program's main logic, yet it should be parted from the main logic. However, aspect in this paper describes an embedded feature in the PoC of existing bugs, which is not explicitly annotated so that we implicitly exploit them by preserving structure and type. Table VIII gives examples of aspects for bugs found by DIE.

In this paper, we advocate a new technique, called *aspect-preserving* mutation, that stochastically preserves beneficial properties and conditions of the original seed input in generating a new test case. We use the term, *aspect* to describe such preferred properties of input corpora that we want to retain across mutation. We claim that the aspect-preserving mutation is a stochastic process because aspects are not explicitly annotated as a part of the input corpus, but are implicitly inferred and maintained by our lightweight mutation strategies. For example, in a PoC exploit, control-flow structures like loops are deliberately chosen to trigger JIT compilation, and certain types are carefully chosen to abuse a vulnerable optimization logic. Under aspect-preserving mutation, we ideally want to maintain with a high chance such aspects in new test cases while introducing enough variations so that we can discover similar or new bugs.

To demonstrate the aspect preservation, we incorporate two new mutation strategies—namely, *structure* and *type* preservation—to our fully-fledged JavaScript fuzzer, called DIE, that implements all modern features like coverage mapping and distributed infrastructure. The foundational technique that enables both mutation strategies is the typed abstract syntax

[1] https://github.com/sslab-gatech/DIE

DOI 10.1109/SP40000.2020.00067

tree, or *typed-AST*, which provides a structural view of an input corpus with the annotated type information of each node. We develop a lightweight type analysis that statically propagates the observed type information extracted from dynamic analysis (§IV-A). Each mutation strategy embodies its own aspect-persevering elements by utilizing the shared typed-AST, *e.g.*, structure-preserving mutation respects structural aspects like loops or breaches, and type-preserving mutation maintains types of each syntactic elements across mutation.

We evaluate DIE with three popular JavaScript engines: ChakraCore in Microsoft Edge, JavaScriptCore in Apple Safari, and V8 in Google Chrome. Our evaluation shows that DIE's aspect-preserving mutation is more effective in discovering new bugs (5.7× more unique crashes) and producing high-quality test cases (2.4× fewer runtime errors) than the state-of-the-art JavaScript fuzzers (see §VI). DIE has newly discovered 48 high-impact security bugs in ChakraCore, JavaScriptCore, and V8;38 of the bugs have been fixed with 12 CVEs assigned as of today ($27K as bug bounty prize).

In summary, this paper makes three contributions:

- We advocate a new *aspect-preserving* mutation approach that aims to preserve desirable properties and preconditions of a seed input across mutation.
- We develop a fully-fledged JavaScript fuzzer, DIE, that implements two new mutation strategies—namely, *structure* and *type* preservation—by using a lightweight static and dynamic type analysis.
- We have reported 48 new bugs and 38 are fixed during the responsible disclosure process: 28 in ChakraCore, 16 in JavascriptCore, and four in V8.

DIE will be open-sourced upon publication.

II. BACKGROUND

In this section, we summarize the common design of JavaScript engines, classify existing fuzzing approaches against them, and analyze a trend of recent JavaScript-related bugs.

A. JavaScript Engines

JavaScript engines are one of the complex components of modern browsers. Although the design and implementation of each JavaScript engine are very different, all share two common properties: 1) serving as a standardized runtime for JavaScript and 2) providing JIT compilation for performance.

JavaScript. This is a dynamically typed language, meaning that a variable can have an arbitrary type at any point during execution. Also, the program can terminate with a syntactic or semantic error at runtime (*e.g.*, invalid references, unexpected types in use). The JavaScript engines process it in multiple phases: a *parser* first creates an AST, and an *interpreter* converts the AST into a first-level intermediate representation (IR) and then executes it with the help of the JavaScript runtime. Note that the parser and interpreter of JavaScript engines have rather simple logics, so no security bugs have been recently reported in either component [13, 14].

JIT compilation. At runtime, the engine profiles execution (*e.g.*, types, # of invocations) to find potential hot spots for optimization. The engine then translates the first-level IR (*i.e.*, bytecode) into a sequence of lower-level IR (*e.g.*, B3 IR in JavaScriptCore) and ultimately to native machine instructions for faster execution. Modern JavaScript engines apply advanced optimization techniques, such as function inlining and redundancy elimination (see Figure 4), during the compilation process. As part of the machine code, the JIT compiler inserts various checks (*e.g.*, types) to validate assumptions made during the optimization, and falls back to the unoptimized code if the optimized code failed at validation, called *bailing out*. Although user-facing interfaces like the parser and the interpreter are the straight implementation of the ECMA262 standard, JIT implementation is specific to each JavaScript engine, *e.g.*, low-level IRs, optimization techniques, etc. In other words, it is a desirable place for security auditing, thanks to the diversity of implementation and the complexity of the optimization logic.

JS Fuzzer	Year	I	T	C	S	D	CVE	OS
jsfunfuzz [39]	2007		G				✓	✓
LangFuzz [20]	2012	✓	M				✓	
Skyfire [43]	2017	✓	M					✓
Fuzzilli [17]	2018		G	✓	✓	✓	✓	✓
CodeAlchemist [19]	2019	✓	G		✓		✓	✓
Superion [44]	2019	✓	M	✓			✓	✓
Nautilus [2]	2019	✓	M	✓				✓
DIE	2019	✓	G/M	✓	✓	✓	✓	✓

I: Input corpus, **T**: Type (G: generative, M: mutational), **C**: Coverage feedback, **S**: Semantic-aware, **D**: Distributed fuzzing, **OS**: Open source

TABLE I: Classification of existing JavaScript engine fuzzers.

B. Fuzzing JavaScript Engines

There are two popular types of JavaScript engine fuzzer, namely, generative and mutational (Table I). Generative fuzzers build new test cases from scratch based on pre-defined grammar like jsfunfuzz [39] and Fuzzilli [17] or by constructing them from synthesizable code bricks disassembled from a large corpus [19]. Mutational fuzzers generate new test cases on the seed inputs for testing. For example, LangFuzz [20] breaks programs in a large corpus into small code fragments, recombines them with a seed input, and generates new test cases; Skyfire [43], Nautilus [2] and Superion [44] mutate each program individually with the segments learned from other programs in the corpus or with their mutation rule. Modern fuzzers [2, 17, 19, 43, 44] all leverage code coverage to accelerate their exploration. However, most advanced generative or mutational fuzzers fail to effectively explore a JavaScript engine for deep bugs on the trend (see §II-A) for two reasons:

1) Enormous search space. One major advantage of generative fuzzers is that they fully control the generation process of every statement in a testing program. Therefore, building error-free inputs is straightforward. However, generative fuzzers build completely new programs by starting from code units. Meanwhile, a JIT-related bug requires a complicated input with specific properties to trigger (see §II-A). Hence, the search space is too large for a generative fuzzer to build such test cases in a reasonable time.

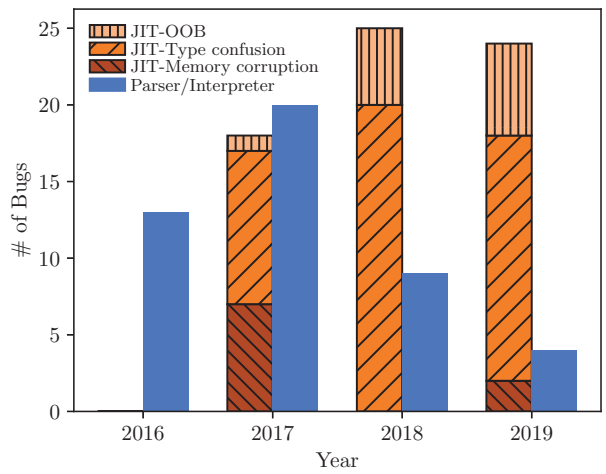

Fig. 1: The trend of the security bugs in ChakraCore from 2016 to 2019. In each column, the right bar shows the number of bugs in ChakraCore's parser, interpreter and its JavaScript runtime. The left bar indicates the number of bugs residing in the JIT compilation phases. We further classify the JIT compiler bugs by their types and out-of-bounds (OOB) access and type confusion caused by incorrect JIT behavior dominate.

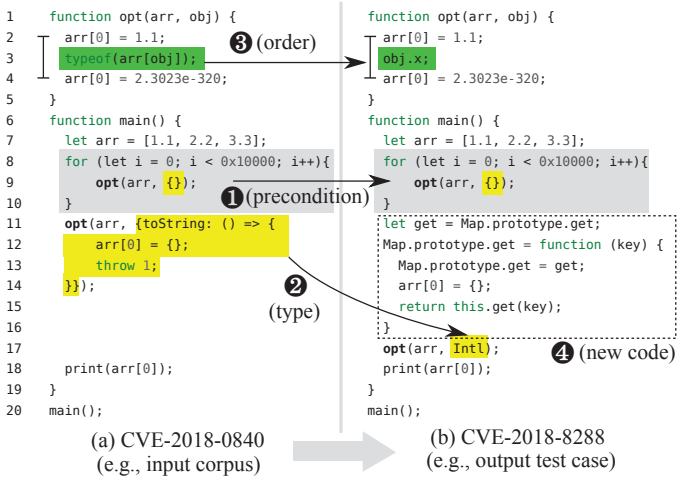

Fig. 2: Two PoC exploits for `CVE-2018-0840` and `CVE-2018-8288` of ChakraCore. Given a PoC-(a) as a seed input, PoC-(b) can only be discovered when three conditions are met (❶–❸) but with enough variation introduced (❹). Note that human hackers are particularly good at exploring similar types of bugs in this manner—both PoC exploits are written by the same author [25, 26].

2) Insufficient utilization of existing programs. Recent JavaScript fuzzers select unit test suites and PoCs of known bugs as their seed inputs. Basically, such a JavaScript program is carefully designed to have certain properties that particularly stress one or more working phases in a JavaScript engine. Therefore, starting with these inputs enables a fuzzer to quickly approach and explore the deep portion of the engine. Unfortunately, existing fuzzers fail to fully leverage this prominent benefit from such programs. For example, the PoC of a JIT-related bug has its unique control flow and data dependencies among used variables, which explore the specific code paths of an optimizer. However, once the PoC is broken into small normalized segments and mixed with the segments of other programs in a large corpus, the generative fuzzers like CodeAlchemist [19] rarely hit the code paths in a reasonable amount of time. Also, semantic aspects are not respected when the PoC is mutated by grammar-rule-based mutational fuzzers like Superion [44] and Nautilus [2]. Different from the aforementioned fuzzers, DIE creates new JavaScript programs in a hybrid manner, synthesizing a seed corpus with a unit generated by generative methods. More importantly, DIE fully respects the properties of unit-test programs and PoCs. In particular, DIE *mutates* an individual program by replacing the code segments that are unrelated to the properties with new ones or simply inserting new statements. Meanwhile, the new segments used for mutation are *generated* from scratch based on the context.

C. Trend of Recent Vulnerabilities

We summarize the vulnerabilities (*i.e.*, exploitable bugs with CVEs assigned) found in ChakraCore from 2016 to 2019 in Figure 1, which demonstrates the trend of vulnerabilities in JavaScript engines. We collect the vulnerability information from Google Project Zero issue trackers [14] and the commits of ChakraCore for security updates [29]. All the vulnerabilities reside in either the parser and interpreter or the JIT compiler at the backend. The number of parser and interpreter bugs has been rapidly decreasing in the period. Meanwhile, the JIT compiler bugs gradually dominate. The JIT compiler bugs are mainly caused by incorrect speculation or wrong optimization based on logic error. We further divide these bugs by their types and notice that most of the bugs result in out-of-bounds (OOB) access or type confusion. An ordinary program written in JavaScript, a typical high-level language, needs sophisticatedly crafted code to trigger these cases. The goal of DIE is to effectively generate such programs that are more likely to hit these deep phases for finding JavaScript bugs in 2019.

III. OVERVIEW

A. Motivation

Human hackers have a particular interest in auditing similar types of vulnerabilities. This intuitively makes sense, as the developers likely introduced similar mistakes dispersedly to the codebase. For example, Figure 2-(a) shows a bug in JIT-related optimizations [25]: a bailout condition is incorrectly assumed and checked in the JIT-ed code. Figure 2-(b) shows a similar bug but with subtle differences: it is incorrectly assumed to have no side effect in an exception (`throw` in `toString`) if suppressed by `typeof` in (a), but in (b), a side effect can be unexpectedly introduced by `Map` used during the initialization of `Intl` (for internationalization support). Accordingly, both exploits share many common preconditions, as shown in Figure 2. For example, ❶ is required to satisfy the JIT-able condition by repeatedly invoking `opt()`, and ❸ is required to trick the optimizer to incorrectly assume the type of `arr` in `opt()` but to allow an optimization condition (*i.e.*, a redundant check elimination) to be met.

Such an intuitive approach is not well respected when a PoC exploit is used as a seed input by automatic techniques like fuzzing. One possible explanation is that the goal of fuzzing by design promotes exploration of new input spaces, which tends to encourage the fuzzer's mutation strategies to *destruct* the conditions encoded in the seeding input. This decision is well made to increase the code coverage in fuzzing small programs.

Nonetheless, the input space for JavaScript-like complex programs having nearly a million lines of source code is infeasible to be exhaustively explored. Also, recent trend of bugs in JavaScript engines is not simple memory corruption bugs that can be achieved by naive coverage-based testing, but rather logic errors that need sophisticated conditions to reach them. In other words, mutation strategies should focus on producing high-quality test cases instead of merely increasing code coverage, so that it can reach meaningful, difficult-to-find bugs. For example, in Figure 2, an ideal strategy for fuzzers is to preserve certain preconditions: keeping the conditions to enable JIT in ❶, a type in ❷ and an access order in ❸, while introducing a new code snippet (❹) that can vary the internal mechanics of JavaScript's optimization algorithms. If Figure 2-(a) is used as an input corpus, existing coverage-based fuzzers likely discard conditions ❶–❸ because they are not essential in discovering new code paths but are considered redundant to the existing corpus.

B. Challenges and Approaches

An ideal fuzzer would want to fully harvest the subtle conditions encoded in a high-quality input corpus such as known PoC exploits [18] or JavaScript unit tests. Thus, the generated test cases naturally satisfy the necessary preconditions to trigger a new bug. One possible approach would be to *manually* encode such conditions in each input corpus. However, this does not work for two reasons: 1) the number of input corpora is huge (14K, see §VI-A), and 2) more critically, it does not provide fuzzers enough freedom for space exploration, negating the benefits of the fuzzing-like automated approaches. Another approach is to *automatically* infer such preconditions from each corpus via program analysis (e.g., data-flow analysis). However, this also negates the true enabler of fuzzers, the performance, i.e., reducing 10% input spaces for exploration after spending 10% more computing power for additional analysis brings no benefit to the fuzzer.

Our key approach is to stochastically preserve *aspects*, the desirable properties we prefer to be maintained across mutation. It is a stochastic process because *aspects* are not explicitly annotated as part of the corpus, but they are implicitly inferred and maintained by our lightweight mutation strategies, so-called *aspect-preserving* mutation.

In this paper, we realize aspect preservation with two mutation strategies, namely, *structure* and *type* preservation:

Structure-preserving mutation. We observe that maintaining certain structures (e.g., control flow) of an input corpus tends to retain their aspects in the generated test cases. For example, a loop structure in a PoC exploit plays a significant role in invoking JIT compilation (❶ in Figure 2), and certain access orders in a JIT-ed region are necessary to trigger an optimization phase (*e.g.*, a redundant check elimination, ❸ in Figure 2). In contrast, widely-deployed *blind* mutation and generation strategies tend to destroy these structures, *e.g.*, at an extreme end, the state-of-the-art JavaScript fuzzer, CodeAlchemist [19], dissects all seeding inputs and regenerates new test cases for fuzzing. According to our evaluation, structure-preserving is the most effective mutation technique to preserve various aspects (*e.g.*, each JIT optimization phases §VI-D) of input corpora across mutation, which renders 2× more crashes than without the technique (Table VII).

Type-preserving mutation. We also observe that respecting types in an input corpus provides enough room for fuzzers to generate high-quality test cases while retaining aspects. For example, an `object` type (❷ in Figure 2) should match with the assumed argument type of the JIT-ed code (`{}` of `opt()` in Line 9), otherwise the code should be bailed out in the JIT execution. In other words, if the types of a seed input are not preserved, the derived test cases end up stressing only a shallow part of the JavaScript engines (e.g., parser or interpreter). In addition, preserving types significantly helps in producing error-free test cases—both syntactic compilation errors and dynamic runtime errors such as `ReferenceError`, `TypeError`, and `RangeError`. Note that such error conditions also prevent test cases from reaching the deep, complex logics of the JavaScript engines, and so they are considered a necessary precondition to preserve various types of aspects of the original seed corpus. In this paper, we leverage a lightweight, type analysis that statically propagates the observed type information from dynamic analysis (§IV-A). According to our evaluation, the type-preserving mutation reduces runtime errors 2× more than the state-of-the-art fuzzer (Figure 5). Note that our type-based mutation also aims to be semantic-aware, meaning that it attempts to avoid the destruction of aspects by respecting some semantics of a seed input, *e.g.*, avoiding `try-catch` logic that thwarts a JIT invocation.

IV. Design

DIE is a mutation-oriented JavaScript fuzzing framework that respects the high-level aspects of seed inputs, such as PoC exploits and unit tests, to generate effective test cases. To achieve this, DIE mutates the AST of a JavaScript input by preserving with a high probability the code structure that affects the overall control flows and the original types of the used variables. Code coverage guidance and a distributed setting also enable DIE to reach deep bugs in JavaScript engines with scale.

Workflow. Figure 3 illustrates DIE's overall design and workflow. First, DIE pre-processes original seed files to produce their *typed ASTs*. Combining dynamic and static type analysis, DIE infers the node types in an AST with low overhead in ①. After type analysis, DIE picks a typed AST of a seed input from the corpus for mutation (② in Figure 3). Given the typed AST, DIE generates a new test case by replacing each node (while preserving its type), or inserting a new node (while preserving the overall structure) (③ in Figure 3). By using the typed AST,

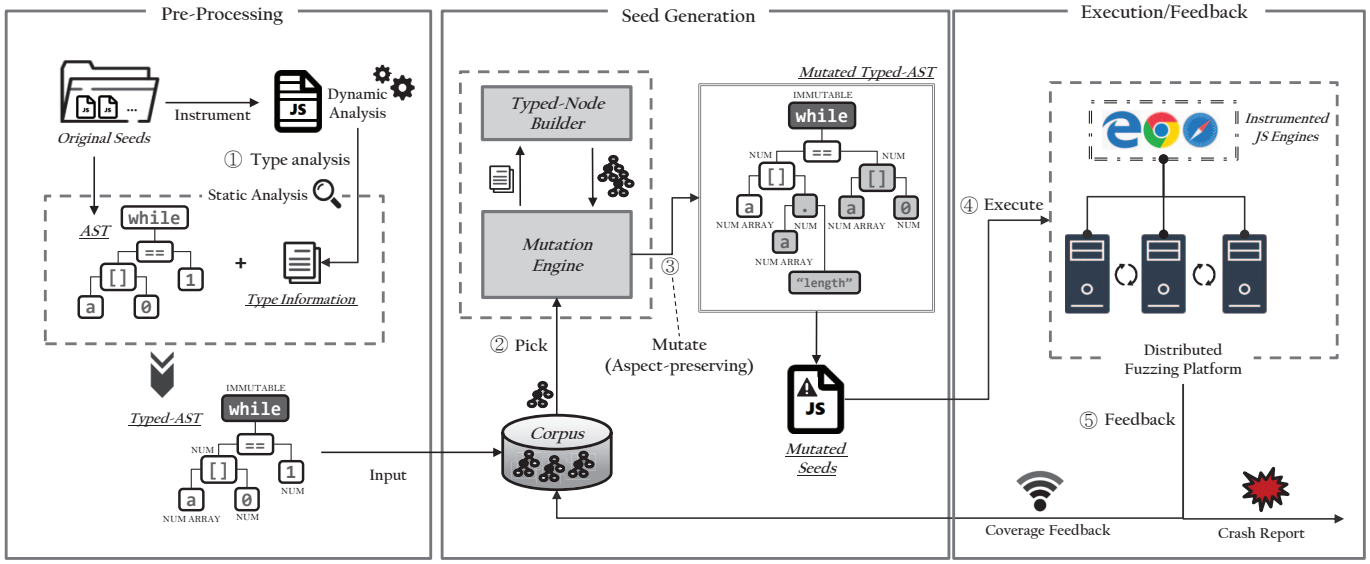

Fig. 3: Design overview of DIE. First, DIE pre-processes (e.g.,instrument) all the original seed files to construct a typed AST via dynamic/static analysis (①). In the main fuzzing loop, DIE selects a test case from the corpus along with its typed AST (②). Next, DIE mutates the typed AST by modifying/inserting new nodes while maintaining its structure and type information (③). Note that the typed-node builder interacts with mutation engine supporting aspect-preserving mutation. Afterward, the AST is converted back to a JavaScript file and examined by the fuzzing platform (④). Finally, DIE measures the runtime coverage for feedback, which determines whether or not the new file will be saved. If the engine crashes at runtime, DIE records the input (⑤). DIE can be deployed in a distributed environment where a certain amount of generated inputs are tested in parallel.

DIE aims to preserve the type during the mutation process, so-called *type-preserving mutation*, and aims to preserve the control-flow structure, so-called *structure-preserving mutation*, each of which tends to maintain certain aspects implicitly embodied in the original corpus across mutation. After mutation, DIE executes the newly generated test case in ④ and checks if the execution crashes or terminates without an error. As the target JavaScript engine is instrumented, DIE can obtain the runtime coverage after executing the test case, and store it as a new input corpus in ⑤ if it visits any new code path. DIE also supports distributed fuzzing by using multiple machines to find bugs concurrently.

A. Custom JavaScript Type System

DIE refines the original type system of JavaScript. The refined type system has two unique properties that are tailored for fuzzing but are different from other JavaScript fuzzers [17, 19, 39]:

Mixed type. DIE introduces a new type called `Mixed` for the syntactic units in a JavaScript program, which captures types that vary at runtime. Note that since JavaScript is a weak- and dynamic- typed language, the type of each variable can only be determined at runtime and even can change during its lifetime. `Mixed` is introduced to describe all types that each syntactic unit can potentially have.

Detailed compound types. DIE inspects the sub-element type(s) of a JavaScript object to define compound types in a more fine-grained manner. (1) `Array`: DIE records the common type of the elements in an array, which can be `Number` or `String`. An array that has empty slots or elements of various types is considered an `Any` array. (2) `Object`: DIE stores the shape of an `Object` instance, which is composed of the types of its property keys and values. (3) `Function`: DIE considers the argument and return type of a `Function` instance.

DIE's custom type system is an essential feature to better support the mutation based on the semantic information extracted from the given test cases. For instance, being aware of several `Number` members of an existing `Object`, DIE has more building blocks for creating a valid `Number` expression than an existing fuzzer. In addition, DIE introduces fewer semantic errors from its mutation. For example, with the refined `Array` types, DIE prefers the element of a `Number` array to an arbitrary array for mutating an array index (*e.g.*, from `arr[i]` to `arr[int_arr[j]]`).

Based on the custom type system, DIE abstracts every JavaScript operation, including expressions and APIs for built-in objects into one or more type reduction rules. Each rule declares the type of the return value when the operation is invoked with the arguments of particular types. Table II summarizes how DIE redefines the *addition* and *array indexing* operations. The default return type of an addition in JavaScript is `String` unless both of the operands are `Numbers`. Moreover, the return value of indexing an array totally depends on the type of element. Note that DIE relies on these rules to infer AST node types for typed AST construction (see §IV-B) and build new AST nodes for a mutation (see §IV-C).

B. Typed ASTs

Basically, DIE mutates the syntactic units (*i.e.*, AST nodes) of a saved JavaScript file in the corpus so as to generate new

Operation	Arg. types	Require l-val.	Ret. type	Ensure l-val.
`arg1 + arg2`	`Num.`, `Num.`	false, false	`Num.`	false
	`Any`, `Any`	false, false	`Str.`	false
`arg1[arg2]`	`Num. Arr.`, `Num.`	false, false	`Num.`	true
	`Str. Arr.`, `Num.`	false, false	`Str.`	true
	`Arr.`, `Num.`	false, false	`Any`	true
`arg1++`	`Num.`	true	`Num.`	false

TABLE II: The examples of typed operation rules described in DIE: addition and array indexing. The rules are used to statically infer the type of every AST node of a seed input and also guide the generation of new typed AST nodes.

inputs for testing. To build new AST nodes for mutation while keeping the validity of a generated input, DIE retrieves the type and binding information of every seed file at the first stage. In particular, for each node of the AST retrieved from a seed file, DIE extends it with (1) its potential type(s) refined by DIE (see §IV-A) at runtime and (2) a set of declared variables available within its scope. We name such an extended AST that contains the type and binding information for every AST node as a *typed* AST. DIE maintains the typed AST for every saved input in the corpus and mutates the typed AST to generate new JavaScript files.

As JavaScript is a weak- and dynamic- typed language, DIE aims to infer all the possible types that an AST node may have at runtime. DIE achieves this through heterogeneous approaches. First, DIE dynamically collects the type(s) of every AST node that represents an identifier, namely, a reference to a particular variable, in a seed file. After parsing out the AST of a seed, DIE instruments the seed by adding a trampoline right before the statement that references an identifier node. The trampoline jumps to a type profiling function that retrieves the type of the identifier at the moment. Note that the function traverses an `Array` and iterates all the members of an `Object` for a refined type enforced by DIE. DIE then runs the instrumented seed file and deduplicates the record types of an identifier output at runtime for an eventual type set. With the determined types of all the leaf nodes (*i.e.*, identifiers and literals), DIE statically infers the type(s) of other AST nodes from bottom to top in an unambiguous manner. Particularly, DIE refers to ECMA-262 [9], which specifies the types of arguments used in a particular expression or built-in API. In addition, DIE statically reducts the types of arguments and return value of a custom `Function` for its legitimate invokings in newly built AST nodes. For the sake of completeness, DIE also labels the statement not having a defined value type with its corresponding descriptive type, such as `if` statements, function declarations, etc. DIE also performs traditional scope analysis for every identifier node in order to build an available variable set at each code point in the typed AST.

C. Building Typed AST Nodes

DIE relies on its builder to build new nodes for mutating the typed AST of an input. Basically, DIE invokes the builder with a desired type and a context (*i.e.*, the code point of a typed AST node where node mutation is to occur). The builder then utilizes the context to construct a new AST whose value type is compatible with the expected one.

Algorithm 1 Constructing a random typed AST that has a desired value type.

```
1:  Input: context: the context of a code point (i.e., bindings)
2:  lval: true only when expecting a l-value
3:  type: the desired value type
4:  step: the current AST depth
5:  Output: the typed AST of a newly constructed expression
6:  function BUILD_EXP(context, lval, type, step = 0)
7:      if step == THRESHOLD then
8:          if PREFER_VAR() or lval then
9:              return BUILD_VAR(context, type)
10:         else
11:             return BUILD_LIT(context, type)
12:     fit_rules ← {}
13:     for each rule ∈ RULES do
14:         if MATCH(rule.ret_type, type) then
15:             if not lval or rule.ret_lval then
16:                 fit_rules ← fit_rules ∪ {rule}
17:     rule ← RANDOM_CHOICE(fit_rules)
18:     args ← {}
19:     for each arg ∈ rule.args do
20:         arg ← BUILD_EXP(context, arg.lval,
21:             arg.type, step + 1)
22:         args ← args ∪ {arg}
23:     return CONSTRUCT_AST(rule, args, rule.ret_type)
```

Algorithm 1 presents the algorithm of the builder that returns the root of a new typed AST. Given a targeted value type, the builder iterates all the abstract rules of the supported JavaScript operations such as the ones described in §IV-A (Line 12-16). The builder randomly selects one that has a matched value type (Line 17) and ensures a l-value if necessary (Line 15). After that, the builder recursively builds the argument nodes required by the operation based on their expected value types (Line 18-22). During the construction, the builder maintains the exact value type for every newly created node during the construction. The builder limits the depth of a new AST and terminates the construction with leaf nodes, including variables and literals (Line 7-11). To fully exploit the semantic aspect of the current seed, the builder tends to reuse the constant values (*e.g.*, numbers, strings, and regular expressions) that appear in the file for building literals; also, the builder references existing variables available within the scope for constructing leaf nodes.

D. Mutating Typed ASTs

Given a selected input, DIE mutates its typed AST in an *aspect-preserving* manner in order to utilize its properties that targetedly test specific code paths in the underlying component of a JavaScript engine. Generally, the aspects of a test case largely depend on its structure and type information. Therefore, during the mutation, DIE particularly avoids removing the entire `if` statements, loop statements, custom function definitions

Algorithm 2 Mutating the typed AST of an input file and testing the mutated input.

```
1:  Input: t_ast: the typed AST of an input file.
2:  procedure FUZZ_ONE(t_ast)
3:      type ← SELECT_MUTATION_TYPE()
4:      if type == Mutation then          ▷ Mutating a typed sub-AST.
5:          old ← RANDOM_EXP(t_ast)
6:          new ← BUILD_EXP(old.context, old.type)
7:          REPLACE_NODE(t_ast, old, new)
8:          SAVE_FILE(t_ast.toString())
9:          REPLACE_NODE(t_ast, new, old)
10:     else
11:         ref ← RANDOM_EXP_STMT(t_ast)
12:         if type == Insert_Statement then   ▷ Inserting a statement.
13:             new ← BUILD_EXP_STMT(ref.context)
14:         else                          ▷ Introducing a new variable.
15:             new ← BUILD_VAR_DECL(ref.context)
16:         INSERT_BEFORE(t_ast, ref, new)
17:         SAVE_FILE(t_ast.toString())
18:         REMOVE_NODE(t_ast, new)
```

and invocations, etc., which determine the structure of an existing JavaScript program. Also, DIE avoids redefining an existing variable with a different type. The mutated AST is then translated into JavaScript code for testing. After processing the execution result, DIE reverts the changes made to the AST for the next round of mutation. If a generated JavaScript file discovers new code paths of the targeted JavaScript engine, its typed AST is saved along with the code. Algorithm 2 presents how DIE fuzzes an existing JavaScript file through aspect-preserving mutation.

Particularly, DIE adopts the following approaches to mutate the typed AST of a JavaScript file, sorted by their priorities:

Mutating a typed sub-AST. DIE randomly selects a sub-AST that serves no structural purpose (*i.e.*, an expression or a sub-expression) (Line 4). The sub-AST is then replaced with a new one built by the builder that has a matched type (Line 5-6).

Inserting a statement. DIE locates a statement block (*e.g.*, the body of an `if` statement, a function or simply the global region) and randomly selects a code point inside the block for insertion (Line 10). Next, DIE generates a new expression statement by using the existing variables declared at the point. Here, the expression statement is simply an expression of any value type followed by a semicolon, which can also be built by the builder. DIE grows the old input by inserting the new statement at the end (Line 15).

Introducing a new variable. Instead of inserting statements, DIE also manages to insert the declarations of new variables at random code points in a typed AST. A new variable is always initialized by an expression built by the builder with a random type (Line 14).

In order to fully exploit the existing aspects of a seed, DIE prefers sub-AST mutation and new statement insertion when selecting the mutation approach (Line 3). DIE introduces new variables into the input only if no new code path is discovered for a long time. The newly generated input through any of the three approaches is stored (Line 7 and 16) for further execution by the target JavaScript engine. Then DIE reverts the changes made to the typed AST that will be reused for further mutation in the subsequent rounds (Line 8 and 17).

E. Feedback-driven Fuzzing in Distributed Environment

Even with DIE's aspect-preserving mutation, finding new bugs in a JavaScript engine is still challenging because (1) input space is still enormous due to the high dimensions of freedom in JavaScript and (2) the re-execution cost is too high to be handled in a single machine. To overcome these issues, DIE uses coverage-driven fuzzing in a distributed environment. In particular, DIE uses classical feedback-driven fuzzing inspired by `AFL`, but with a refined code coverage that is the same as Fuzzilli [17]. The original AFL's code coverage represents each edge in a byte to record hit counts for finding overflows. However, hit counts are pointless in JavaScript because they can be arbitrarily controllable by modifying a range in the `for` statement. Thus, DIE records an edge in a bit by discarding hit counts. As a result, DIE can store eight times more branches within the same size of memory compared to the original AFL's design.

Furthermore, to make multiple machines collaborate in a distributed environment, DIE develops its own code-coverage synchronization mechanism. While maintaining its local coverage map, an instance of DIE synchronizes its map with a global map if it discovers a locally interesting test case; it introduces a new bit according to the instance's local map. Then, the instance uploads the test case if it is still interesting after the synchronization. This is similar to EnFuzz's Globally Asynchronous Locally Synchronous (GALS) mechanism [6], but is different in two aspects. First, DIE synchronizes code coverage itself instead of interesting test cases like EnFuzz. Unlike EnFuzz, which needs to re-evaluate test cases because of the heterogeneity of fuzzers, DIE can avoid this re-execution, which is expensive in JavaScript engines, by synchronizing code coverage directly thanks to identical fuzzers. Second, DIE can support multiple machines, not just multiple processes in a single machine.

V. IMPLEMENTATION

Broadly, DIE is implemented as an AFL variant that can also run in a distributed environment. First, DIE introduces a pre-processing phase into AFL. Starting with existing test suites and PoCs, DIE leverages its *type analyzer* to construct their typed ASTs and save them along with the source files into the input corpus. More importantly, DIE replaces the AFL's mutator for binary input with its own mutation engine. The mutation engine uses the typed-AST builder to build random sub-ASTs for mutating or growing a typed-AST selected from the corpus. DIE reuses most of the other components of the

Component	LoC	Language
Fuzzing engine		
Type analyzer	3,677	TypeScript
Instrumentation tool	222	Python
Typed-node builder	10,545	TypeScript
Mutation engine	2,333	TypeScript
AFL modification	453	C
Distributed fuzzing harness		
Coordinator	205	TypeScript
Local agent	1,419	Python, Shell script
Crash reporter	492	Python

TABLE III: Implementation complexity of DIE including core fuzzing engine, AFL modification, and necessary harnesses for distributed fuzzing. Since we reuse a few components (*e.g.*, the fork-server, seed scheduler, and coverage collector) in the original AFL, we omit their code sizes.

AFL, including the fork-server, seed scheduler, and coverage collector. Nevertheless, DIE disables the trimming phase in the original AFL, which destructs the aspects from a seed input. Note that DIE heavily utilizes Babel [28], a popular JavaScript transpiler, to complete all the tasks at the AST level of a JavaScript file. To support DIE to run in a distributed manner, we implement several harnesses: (1) a centralized coordinator that synchronizes discovered test cases across the DIE instances running on different machines and collects crashes, (2) a local agent that manages the execution of DIE on a single machine, and (3) a crash reporter that deduplicates found crashes and reports the coordinator.

Table III presents the lines of code (LoC) of each component of DIE. We explain the implementation details of several non-trivial ones in this section.

Type analyzer. The type analyzer constructs a corresponding typed-AST for every seed file (see §IV-B). First, it leverages Babel to get the original AST of a seed file. Note that Babel also provides the binding information of every variable along with the AST by scope analysis. Then it instruments the seed file with the invokings of the typing function on every occurrence of a variable (*i.e.*, an identifier) with the help of the Babel APIs for AST manipulation. Then the seed file is executed and the runtime types of an identifier are collected at runtime and used for bottom-up type analysis on the AST afterward. To implement a typed-AST, we simply introduce a new `type` field into the original `Node` structure that represents a node in the AST implementation of Babel, where the inferred type is stored.

Mutation engine and typed-node builder. Given an input typed-AST from the corpus, the mutation engine queries the typed-AST builder for a randomly built sub-AST. The built sub-AST is a substitute for an existing subtree or is simply inserted into the input typed-AST (see §IV-D). Since our typed-AST is a simple extension of the Babel's AST, we leverage various Babel APIs for (1) building new ASTs of JavaScript expressions and statements in the typed-node builder and (2) removing existing nodes or adding new nodes in an AST in the mutation engine. Also, we gather the literals (*i.e.*, numbers, strings and regular expressions) used in all the seed programs in our corpus for the builder to choose from when building a new code segment. The builder also works with the binding information provided by the original AST in Babel and thereby only uses declared variables within the scope in order to avoid `ReferenceError`.

Integration with AFL. We build a single fuzzing executor of DIE on the basic infrastructure of AFL [49] (version 2.52b), including the forkserver, coverage feedback routine, and execution scheduling. Instead of classic byte mutation in AFL, we integrate our fuzzing engine for mutation with AFL and let it communicate with the AFL infrastructure. After generating input by the core engine, AFL executes a target engine with the generated input. Libraries, including wrapper functions to resolve compatibility issues (see §VI-A), are executed together with the generated input. To build an instrumented binary for code coverage, we directly reuse AFL-Clang as a compiler and slightly modify the LLVM pass to use the custom code coverage described in §IV-E.

Distributed fuzzing engine. To launch our fuzzing executor in a distributed environment, we implement harnesses, including executor written in Python and Shell script, coordinator, and crash reporter. To execute the shell command in a distributed environment, we use fabric [10], which is a Python library supporting the execution of shell commands remotely. The harness includes functionalities such as installing dependencies and deploying, launching, and terminating the AFL instances. After launching them in a distributed environment, diverse intermediate data such as seed corpora should be synchronized. For fast and reliable data access in a distributed environment, we use a well-known open source and in-memory database, redis [35]. The coordinator communicates with the redis server and synchronizes and distributes intermediate data (*e.g.*, code coverage and input introduced new paths) among the distributed AFL instances. We also implement crash reporter to report the found crashes and filter them to eliminate duplicates. Once the crash reporter gets crashes from the database, it tests them with the JavaScript engine and notifies the user if it finds a new unique crash.

VI. EVALUATION

In this section, we evaluate the effectiveness of DIE regarding its ability to find new bugs in the latest JavaScript engines using aspect-preserving mutation. Moreover, we compare DIE with existing JavaScript engine fuzzers based on diverse metrics to present various advantages of DIE.

- **Q1** Can DIE find new bugs in real-world JavaScript engines? (§VI-B)
- **Q2** Do the preserved aspects from the corpus play a key role in triggering the bugs found by DIE? (§VI-C)
- **Q3** Does DIE fully preserve the aspects presented by the corpus? (§VI-D)
- **Q4** Can DIE generate correct JavaScript code, both syntactically and semantically? (§VI-E)
- **Q5** How does DIE perform in terms of code coverage and bug finding ability against state-of-the-art fuzzers? (§VI-F)

§VI-A explains the environment for the experiments. §VI-B describes the bugs, including security vulnerabilities found by DIE. §VI-C evaluates the effectiveness of utilizing the aspects of the seed corpus by analyzing the results of DIE. §VI-D evaluates whether the aspects of the seed corpus are well maintained in the test cases generated by DIE. §VI-E evaluates the validity of the generated input by DIE based on syntactic and semantic errors raised by JavaScript engines. §VI-F compares the performance of DIE with other state-of-the-art fuzzers.

A. Experimental Setup

Environment. We evaluate DIE on Intel Xeon E7-4820 (64 cores) with 132 GB memory for the experiments in §VI-D and §VI-F, and Intel Xeon Gold 5115 (40 cores) with 196 GB memory for the ones in §VI-E. Both machines run Ubuntu 16.04. Note that when compared with other fuzzers that do not natively support distributed fuzzing, we only use a single machine in the evaluation for fairness.

Targeted engines. We evaluate the bug-finding ability with three widely used JavaScript engines: ChakraCore [29], JavaScriptCore [1], and V8 [16]. Note that these engines currently operate for Microsoft Edge, Apple Safari, and Google Chrome, which all have a large user base and are security-critical so that they are heavily tested by OSS-Fuzz [15] and security researchers. Also, we choose the youngest engine, ChakraCore, as a representative in the other experiments (*i.e.*, evaluating aspect preserving, input validity, and code coverage). With a design similar to other engines, ChakraCore involves abundant complicated compiler techniques for code optimization and also provides fine-grained debug messages in each working phase.

Collecting valid seed inputs. As DIE mutates based on the aspects of existing test suites and PoCs, the quality and validity of seed corpora largely affect DIE's performance. To build the corpus of DIE, we collect JavaScript files from two public sources: (1) regression tests from the source repositories of four JavaScript engines: ChakraCore, JavaScriptCore, V8, and SpiderMonkey, and (2) js-vuln-db [18], a public repository that collects PoCs of JavaScript engine CVEs. To alleviate compatibility issues among different JavaScript engines, we clarified the engine-specific functions (*e.g.*, Windows Script Host (WScript) in ChakraCore) and then implemented wrapper functions that perform the equivalent actions in the other engines or eliminated them as possible to suppress unexpected `ReferenceError`. Moreover, to fully utilize the seed corpus, we further removed all the assertions from the collected files to prevent early termination of the new inputs generated by DIE. We eventually accumulated 14,708 unique JavaScript files, including 158 JavaScript files from js-vuln-db used in the following experiments [2].

[2]We use complete set of files from these repositories to avoid cherry-picking or biased selection.

JS Engine	Version	# Lines	Running Time	Resource
ChakraCore	1.11.5	780,954	3 days	N: 22, C: 839
ChakraCore	1.11.5*	797,872	3 days	N: 22, C: 839
ChakraCore	1.11.9	781,397	1 week	N: 22, C: 839
ChakraCore	1.11.9*	797,782	1 week	N: 22, C: 839
JavaScriptCore	2.24.2	443,692	1 week	N: 22, C: 839
V8	8.0.0*	995,299	1 week	N: 13, C: 388

N: # of nodes, C: # of cores
*Canary version

TABLE IV: Targeted JavaScript engines, their versions and the running time DIE runs against them.

Preserved aspect	Bug	Crash
Structure & Type	14/28 (50.00%)	40/84 (47.62%)
Structure-only	12/28 (42.86%)	32/84 (42.86%)
Total	22/28 (92.86%)	72/84 (90.48%)

TABLE V: The ratio of the crashes and bugs found by DIE in ChakraCore that exactly borrow the aspects, indicated by both structure and type information, or only the control flow structure, from the seed files in the starting corpus.

B. Identified Bugs Including Security Vulnerabilities

To evaluate the ability of DIE in finding new vulnerabilities, we comprehensively ran DIE in a distributed environment, including one master node to store and synchronize intermediate data (*e.g.*, coverage map) and multiple slave nodes. Table IV describes the targeted engines, period, and used resource DIE ran for.

As a result, DIE found 28 bugs in ChakraCore, 16 bugs in JavaScriptCore, and four bugs in V8 for a total of 48 bugs. Table VIII shows the unique bugs found and their description. We counted these bugs using the following criteria: (1) found but fixed issues before we reported to the vendors, (2) semantic bugs that have different behavior from spec and other JavaScript engines, (3) memory corruption bugs except assertions in release build, and (4) security bugs acknowledged by vendors. Actually, assertion in release build can be considered a type of bug for some vendors. For example, the vendor of JavaScriptCore accepts reports related to assertions in release build and was willing to fix them, although they are not security-related bugs. On the other hand, the vendor of ChakraCore does not accept reports about assertion in release build. Thus, we eliminated the number of assertions in release build to conservatively count the number of bugs found.

To identify all bugs by unique root cause, we manually analyzed every found crash, and identified 48 distinct bugs. Among the distinct bugs, we found that 16 are related to security based on their similarity to existing bugs previously known as security-related bugs. Of the number of security-related bugs, we gained 12 CVEs acknowledged by vendors and 27K USD as bug bounty rewards. In addition, 13 of the bugs are likely security-related, including memory corruption bugs. Interestingly, we could identify six semantic bugs in ChakraCore because ChakraCore provides a more detailed debugging message than the others for misbehaved situations including semantic bugs, so DIE could reach them.

```
1  function opt(arr, start, end) {
2    for (let i = start; i < end; i++) {
3      if (i === 10) {
4        i += 0;
5      }
6  +   start++;
7  +   ++start;
8  +   --start;
9      arr[i] = 2.3023e-320;
10   }
11 + arr[start] = 2.3023e-320;
12 }
13 function main() {
14   let arr = new Array(100);
15   arr.fill(1.1);
16
17   for (let i = 0; i < 1000; i++) {
18 -   opt(arr, 0, 3);
19 +   opt(arr, 0, i);
20   }
21   opt(arr, 0, 100000);
22 }
23 main();
```

Listing 1: The difference between the PoC of CVE-2019-0990 found by DIE and that of CVE-2018-0777 contained in the corpus. The PoC is almost seemingly identical, yet patching these bugs requires different measure as their root cause differs from each other.

C. Effectiveness of Leveraging Aspect

As described in §III-B, DIE leverages the aspects from existing test cases to explore a broad input space more efficiently and effectively. To evaluate whether aspect-preserving mutation enables us to reach bugs, we manually investigated the relationship between the generated crashing inputs in §VI-B and their corresponding seed files. First, we minimized every crashing input into a minimal PoC that can trigger the crash. We then inspect whether the structure or type information of the PoC that result in the crash correspond to that of the seed file indeed. We checked the inputs for 84 distinct crashes and 28 reasoned bugs found by DIE in ChakraCore (see §VI-B). Table V presents the number of inputs that leverage only structure information or both structure and type information of the original seed file. The result shows that the aspects borrowed from the starting corpus contribute to 90.48% of the crashes and 92.86% of the bugs found by DIE. In particular, 47.62% of the crashes and half of the bugs share both structure and type information with the corpus. The detailed aspects of the found bugs are described in Table VIII.

Listing 1 presents a code difference between a bug found by DIE (*i.e.*, CVE-2019-0990) and its seed file (*i.e.*, CVE-2018-0777) to show an example of shared aspects. The original seed corpus leads to an out-of-bounds array access (Line 9), as the JavaScript engine fails to compute the correct bounds of the array (*i.e.*, *arr*), so a bound check for the array is incorrectly eliminated by redundancy elimination for optimization. This is because the array index created as an induction variable (*i.e.*, *i*) is wrongly optimized (Line 4). Similar to the seed, the bug found by DIE leads to an out-of-bounds array access (Line 11) due to a wrong bound check elimination for the array. It also uses an induction variable (*i.e.*, *start*) as an array index and it is wrongly calculated [3], so it leads to a miscalculated array bound that affects the wrong array-bound elimination. This example shows the benefit of DIE in terms of borrowing existing aspects, wrongly calculated induction variable, and using it as an array index to invoke a wrong redundancy elimination. The structure-preserving supported by DIE helps to keep the environment, which leads to a wrong redundancy elimination (*e.g.*, *for* and *if* statement on Line 2-5), and type-preserving mutation (*e.g.*, *i* on Line 19) helps it to iterate over the loop enough times to lead to wrong induction variable calculation. Note that the bug is not reproducible if the *if* statement (Line 3-5) is eliminated, which means a negligible code change affects the optimization phase in JIT sensitively, which leads to the bug. Besides, although the difference between the two PoCs seems trivial, their root cause differs; thus, patching these two bugs requires independent effort. The root cause of the previous bug (CVE-2018-0777) stems from erroneous constant folding, whereas the new bug brings its wrong behavior due to the improper array bounds profiling.

[3]Patch: https://bit.ly/2MEahCK

	DIE	DIE$_t$	Superion	CodeAlchemist
Preserved aspect	65.39%	34.26%	58.26%	40.67%
# of bytecode†	412	1,422	-119	-984

† The corpus totally emits 2,551 unique bytecode statements after normalization.

TABLE VI: The preserved aspect rate of generated input and the difference between the number of normalized statements in the bytecode of the seed programs and the generated inputs by DIE, DIE$_t$, Superion, and CodeAlchemist.

D. Evaluation of Aspect Preserving

To demonstrate that preserving a structure and type information are effective to maintain interesting aspects and compare the performance regarding aspect preserving with existing fuzzers, we evaluate DIE, DIE without structure-preserving, Superion, and CodeAlchemist with a seed corpus that only contains the JavaScript programs that triggers JIT compilation. Note that the approach of DIE without structure-preserving (notated as **DIE$_t$** for convenience) mutates any node in a typed AST regardless of the node's structural meaning. DIE$_t$ still respects the type information during its mutation. First, we measure the rate of generated input invoking the JIT compilation, which is considered a criterion to show aspect-preserving. Next, we compare the number of unique (normalized) statements in the emitted bytecode of the generated input with the number in the seed corpus to further demonstrate the power of aspect-preserving mutation in exploiting existing test cases and covering deep code paths in a JavaScript engine. When counting statements in the bytecode, we normalize the operands (*e.g.*, literal and register name in arguments of bytecode) that are false noises that hinder the true uniqueness. Last, in order to show a more fine-grained effect of preserving a structure and type information for utilizing the aspects of an existing test case, we evaluate the ratio difference of JIT-optimization invocations between the set of generated inputs and the starting corpus. Note that we choose Superion and CodeAlchemist for comparison, as they are one of the

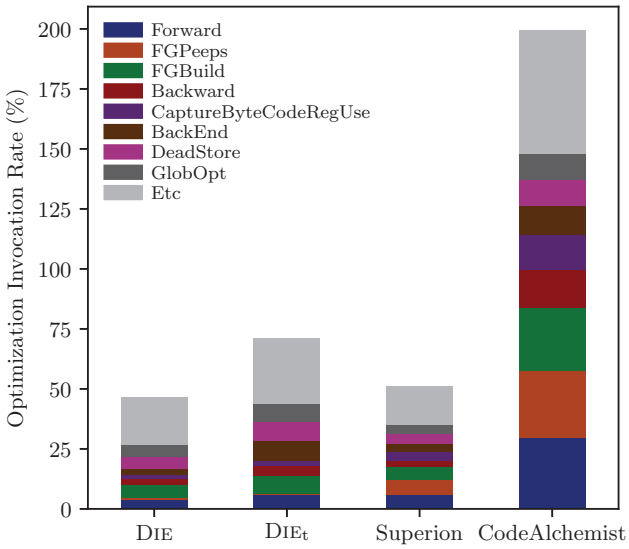

Fig. 4: The ratio difference of JIT-optimization invocations between the generated inputs and seed files. The y-axis represents the absolute difference between the rate, which means the number of invoked optimization phases per JIT invocation. Based on the same seed corpus, DIE is the least distant from the corpus regarding optimization invocations. The eight most notable optimization phases are highlighted.

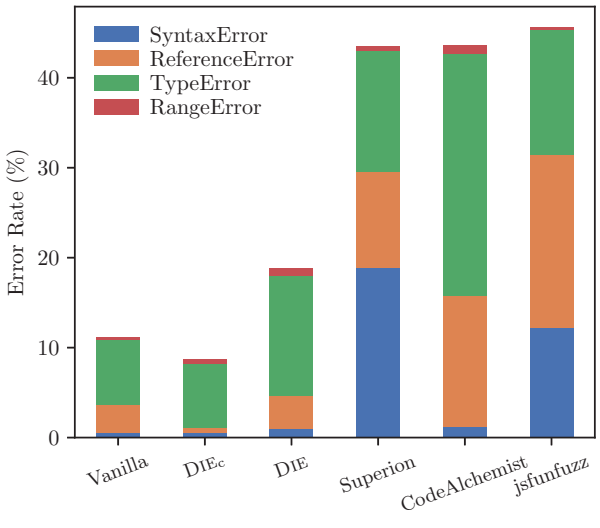

Fig. 5: The error rate of seed files and inputs generated by DIE without coverage feedback (*i.e.*, **DIE$_c$**), DIE, Superion, CodeAlchemist, and jsfunfuzz against ChakraCore for 12 hours. The y-axis represents the rate of the generated inputs yielding runtime errors. Based on the same seed corpus, both Superion and CodeAlchemist generate 2.31× more runtime errors than DIE. In the meantime, jsfunfuzz generates 2.42× more runtime errors than DIE.

state-of-the-art JavaScript fuzzers that found JavaScript bugs that are acknowledged as CVEs and uses the corpus for input mutation and generation.

Aspect preserving presented as JIT invocations. We observe in Table VI that DIE generates 1.12× and 1.61× more inputs than Superion and CodeAlchemist that invokes JIT compilation. The result shows that Superion also tends to maintain aspect through mutation, but DIE is better than Superion because our approach is aware of not only grammar but also semantics for mutation. In addition, the approach of CodeAlchemist is far from leveraging aspects because it breaks useful aspects in the seed corpus while breaking corpora into code bricks. Furthermore, comparison between DIE and DIE$_t$ shows the importance of maintaining the structure in regard to aspect preserving. Specifically, DIE invokes JIT compilation 1.9× more times in the experiment.

Aspect preserving presented as optimization invocation. To demonstrate the approach that DIE helps to preserve aspect in a more fine-grained way, we measured the number of invoked optimization phases and compared it with the one invoked by the files in the seed corpus. Figure 4 shows that DIE modifies the fewest aspects in the seed against DIE$_t$, Superion, and CodeAlchemist. DIE maintains the same optimization invocation 1.53× and 4.29× more than DIE$_t$ and CodeAlchemist, respectively. In particular, CodeAlchemist makes the biggest difference for every optimization phase, which means reassembling code bricks can totally break the existing aspects in the seed corpus. Superion shows the negligible difference from DIE because Superion fuzzes code in a dumb manner, which is proven as it generates less diverse bytecodes (see Table V) and higher syntax error (see Figure 5). However, the dumb manner leaves JIT-affected code in the seed corpus intact in many cases, so it does not hurt the optimization invocation a lot.

Unique bytecode generation. DIE generates more diverse bytecodes than the ones emitted by the seed corpus whether the structure is preserved or not (see Table V). In contrast to DIE, Superion and CodeAlchemist produce less diverse bytecodes than the ones of the seed corpus, which indicates that both fuzzers cannot fully utilize existing test cases. Also, the result shows that DIE tends to explore the input space in a more diverse way than the others.

E. Validity of Generated Input

As §III-B described, generating valid highly-structured input is difficult but important because early termination by invalid input will hinder further exploring the input space, which may include defects. To answer **Q4**, we measured the runtime error rates while executing generated input by Superion [44], CodeAlchemist [19], jsfunfuzz [39], and DIE. We slightly modified the fuzzers to check the standard error streams of executing the generated input with the JavaScript engine to compare DIE against existing fuzzers.

[3]To confirm the fairness of comparison, we conducted manual inspection of root causes of high error rate generated by CodeAlchemist. We observed that high error rate majorly stems from incorrect variable handling (i.e., redeclare existing variables and redefine them with wrong types) between assembled code bricks.

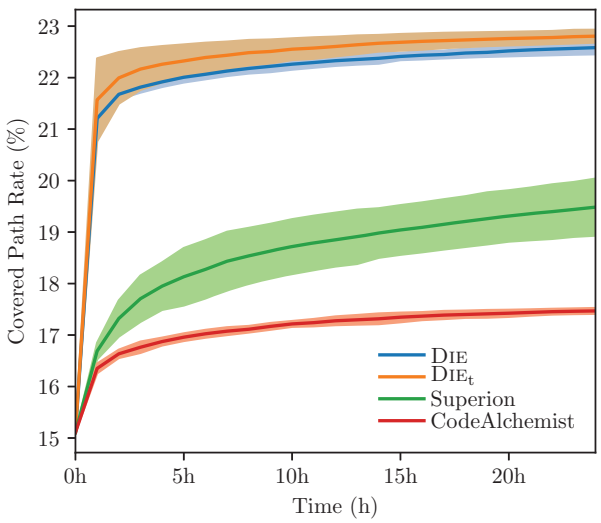

Fig. 6: The overall covered path rate of running DIE, DIE without structure-preserving (*i.e.*, DIE$_t$), Superion, and CodeAlchemist against ChakraCore for 24 hours. The solid lines represent mean and the shades around lines are confidence intervals for five runs. In the process, DIE visits unique paths up to 1.16× more than Superion and up to 1.29× more than CodeAlchemist. The unique paths visited by DIE$_t$ slightly exceed the original DIE up to 1.01×.

JavaScript Engine	DIE	DIE$_t$	Superion	CodeAlchemist
ChakraCore 1.11.10	17	7	0	3
JavaScriptCore 2.24.2	2	0	0	0
V8 7.7.100	2	1	1	0

TABLE VII: The number of unique crashes found by DIE, Superion and CodeAlchemist for 24 hours on the latest engines of June, 2019. All of the crashes were manually inspected to confirm its uniqueness.

Figure 5 presents the error rate of generated input by DIE and existing fuzzers. First, we measured the error rate of the intact seed files (*i.e.*, vanilla) for comparison. The set of seed files originally generated an 11.20% error rate when we performed a dry-run. With the seed set, the newly created inputs by DIE produced an 18.88% error rate with coverage feedback. In addition, Superion and CodeAlchemist generated a 43.54% and 43.58% error rate with the same seed set, and jsfunfuzz generated a 45.62% error rate, which means that the generated inputs by both Superion and CodeAlchemist yield 2.31× more runtime error [4]and generated inputs by jsfunfuzz yield 2.42× more runtime error than DIE. This result demonstrates the effectiveness of DIE against existing JavaScript fuzzers with regard to generating valid input, which is an important factor to a fuzz structured target. Furthermore, we measured DIE without coverage feedback to show the ability of DIE to construct valid code. Without coverage feedback, DIE generated an 8.65% error rate, which is less than the error rate of the original seed set. The result indicates that DIE analyzes type correctly, builds a valid typed AST node based on our type system, and replaces the error-yielding AST node with the valid AST node.

F. Performance Comparison with Other Fuzzers

To compare performance in terms of exploring input space and reaching crashes against state-of-the-art fuzzers, we first ran DIE, DIE$_t$, Superion, and CodeAlchemist on the instrumented JavaScript engine to measure code coverage. In addition, we ran them in the same environment (see §VI-A) against the latest versions of three major JavaScript engines and counted the number of crashes they found. Both experiments lasted for 24 hours.

Exploring input space. Figure 6 illustrates that DIE explores unique code paths up to 1.16× more than Superion and 1.29× more than CodeAlchemist. Interestingly, DIE$_t$ visits slightly more paths than the original DIE. This is because DIE$_t$ has more chances to mutate diverse nodes in a more diverse manner, which matches the result in Table V that it produces more unique bytecodes.

Reaching crashes. Table VII summarizes the number of unique crashes each fuzzer found. Note that DIE found the most unique crashes [5] on the JavaScript engine, while Superion and CodeAlchemist found fewer crashes. In particular, DIE found 5.7× more than CodeAlchemist on the latest version of ChakraCore, and Superion could not find any crash over the same period.

From the result, we observe that most code paths were introduced within the first two hours, which shows that leveraging aspects in the seed helps to boost exploring diverse paths. More importantly, we conclude that code coverage tends to show the ability of input-space searching, but it cannot be the absolute metric to judge the ability of a JavaScript engine fuzzer to find bugs: (1) DIE$_t$ introduced more code coverage than DIE, but found fewer crashes, and (2) 71.79% of the crashes found by DIE in ChakraCore are generated after the first two hours.

VII. Discussion

We have demonstrated that DIE effectively leverages the *aspects* to discover bugs in the latest JavaScript engines. In this section, we discuss the limitations of DIE and our future directions.

Seed prioritization. The mutation approach of DIE highly relies on the seed files, which means the quality of the starting corpus is an important factor that determines the result. DIE currently does not prioritize seeds for mutation, which may make DIE waste time on mutating the seed files that do not have valuable aspects and discourage DIE from exploring faceted ones. We believe DIE benefits from the state-of-the-art seed selection algorithms [34, 43].

Generative rule-based builder. DIE generates typed nodes based on the language rules. In particular, DIE includes most operations allowed in JavaScript to generate diverse code segments. However, DIE uniformly selects the generation rules to build new nodes. Researchers can prioritize certain rules to heavily test specific routines in JavaScript engines. Moreover,

[5]All crashes are manually inspected, suggested by [22], to confirm their distinctness instead of AFL's coverage measure.

DIE can integrate several existing approaches to generate new code such as utilizing code fragments [19, 20] or IR-based generation [17].

Aspect annotation. Practically, DIE considers that the structure and type information of a test case form its aspects when fuzzing JavaScript files. Basically, this information is feasible and largely affects how a JavaScript engine JIT-compiles and further optimizes a program. Nevertheless, one is free to annotate the aspects of a seed file [30, 32] with different semantic information and explore more specific code paths.

Aspect-preserving mutation beyond JavaScript. Although DIE only focuses on fuzzing JavaScript engines now, the concept of aspect-preserving mutation is generic enough to be applied against any target. First, the core idea of DIE can be ported to other language compilers or interpreters for other contexts.

For example, Equivalence Modulo Inputs (EMI) [23, 24] is proposed to validate optimizing compilers for differential testing. Similar to DIE, it utilizes existing input corpora to construct valid test programs. In addition, it selectively mutates unexecuted code to fully exploit the existing semantics that can correspond to the aspect of DIE. We believe that the mutation algorithm of these works can benefit from DIE by cooperating with its type-preserving mutation based on structure preserving. This is technically doable in C context with a mutation skeleton [5]. Also, [45] proposes a marking algorithm, which has a similar effect to preserve structure in DIE, to fuzz an ActionScript virtual machine. Instead of marking only the identifier, our type-preserving mutation can help to improve the marking algorithm in that context as well.

Furthermore, we can adopt the concept of aspects for applications that receive binary input by identifying the aspects of a seed file. For instance, using taint analysis can deprioritize the modification of certain bytes of the file based upon the analysis results. The notable bytes contribute to the aspects.

VIII. Related Work

Syntax-aware fuzzing. The earliest fuzzers for structured inputs worked for being aware of their syntax [3, 11, 41, 46–48]. In JavaScript fuzzing, jsfunfuzz [39], and LangFuzz [20] are frontiers in this line of work. jsfunfuzz generates various JavaScript programs from scratch based on its pre-defined rules, while LangFuzz modifies existing test cases by randomly combining their code fragments. Unlike these approaches, DIE considers not only syntax but also semantics to generate test cases with fewer runtime errors.

Semantic-aware fuzzing. After proposing a line of syntax-aware JavaScript engine fuzzers, researchers have started to build semantic-aware ones [19, 31, 42, 43] for better performance. Skyfire [43] is one of the earliest research efforts that tackles the semantic problem in language fuzzing. Skyfire learns the semantics of a language from existing test cases in the form of probabilistic context-sensitive grammar (PCSG), which is further used for fuzzing. Unlike Skyfire, CodeAlchemist [19] focuses more on correctly using variables in the generated code based on their types so as to create more semantically valid inputs. Machine learning is also applied to master sophisticated semantic rules from numerous seed files [8, 12], which are used for generating more test cases. Not only aimed at building semantically correct inputs like these works, DIE targetedly stresses specific components in a JavaScript engine by fully utilizing the overall semantic properties of each existing test case (*i.e.*, aspects).

Coverage-guided fuzzers. Starting from AFL [49], coverage-guided fuzzing became very popular among general-purpose fuzzers [7, 27, 36, 38] and also for JavaScript fuzzing [2, 44]. Superion [44] extends AFL to support additional mutation strategies for grammar-based inputs such as XML and JavaScript. As a result, Superion benefits from coverage feedback by better preserving the structure of a JavaScript program. Similarly, Nautilus [2] leverages coverage feedback with context-free-grammar-based input generation. Fuzzilli [17], a recently introduced generative JavaScript fuzzer, also relies on coverage feedback. Based on a specially designed Intermediate Representation (IR), Fuzzilli builds syntactically and semantically correct test cases from scratch. Different from these approaches that arbitrarily modify test cases only for maximizing code coverage, DIE limits its mutation for aspects to meet the complex conditions of modern JavaScript bugs.

IX. Conclusion

In this paper, we propose DIE, a JavaScript engine fuzzer that preserves the aspects of a pre-mutated test case, which are the essential conditions for its original purpose. To this end, DIE deliberately handles the structure of a given test case and keeps its type information intact using our novel type analysis in a static and dynamic manner. Our evaluation shows that DIE can maintain 1.61× more aspects than state-of-the-art fuzzers, including Superion and CodeAlchemist, resulting in 5.7× more unique crashes. More importantly, DIE found 48 new bugs in real-world JavaScript engines with 12 CVEs assigned.

X. Acknowledgment

We thank the anonymous reviewers, and Frank Piessens especially, for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE, and ETRI IITP/KEIT[2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware and Google.

References

[1] Apple. JavaScriptCore, The built-in JavaScript engine for WebKit, 2019. https://trac.webkit.org/wiki/JavaScriptCore.

[2] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[3] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing program input grammars. In *ACM SIGPLAN Notices*, volume 52, pages 95–110. ACM, 2017.

[4] O. Chang, A. Arya, and J. Armour. OSS-Fuzz: Five Months Later, and Rewarding Projects, 2018. https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html.

[5] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 27th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Tallinn, Estonia, Aug. 2019.

[6] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, Z. Su, and X. Jiao. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, USA, Aug. 2019.

[7] N. Coppik, O. Schwahn, and N. Suri. Memfuzz: Using memory accesses to guide fuzzing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 48–58. IEEE, 2019.

[8] C. Cummins, P. Petoumenos, A. Murray, and H. Leather. Compiler fuzzing through deep learning. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Amsterdam, Netherlands, July 2018.

[9] ECMA. Standard ECMA-262. https://www.ecma-international.org/publications/standards/Ecma-262.htm, 2019.

[10] J. Forcier. Fabric, High level python library designed to execute shell commands remotely over SSH, 2019. http://www.fabfile.org/.

[11] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Tucson, Arizona, June 2007.

[12] P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana-Champaign, Illinois, USA, Nov. 2017.

[13] Google. Chrome Releases. https://chromereleases.googleblog.com, 2019.

[14] Google. project-zero. https://bugs.chromium.org/p/project-zero/issues/list, 2019.

[15] Google. Continuous fuzzing of open source software, 2019. https://opensource.google/projects/oss-fuzz.

[16] Google. V8, Open source JavaScript and WebAssembly engine for Chrome and Node.js, 2019. https://v8.dev/.

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

[18] C. Han. js-vuln-db, A collection of JavaScript engine CVEs with PoCs, 2019. https://github.com/tunz/js-vuln-db.

[19] H. Han, D. Oh, and S. K. Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[20] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[21] Honggfuzz. Honggfuzz Found Bugs, 2018. https://github.com/google/honggfuzz#trophies.

[22] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[23] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, United Kingdom, June 2014.

[24] V. Le, C. Sun, and Z. Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 26th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Pittsburgh, PA, Oct. 2015.

[25] J. Lee. Issue 1438: Microsoft Edge: Chakra: JIT: ImplicitCallFlags checks bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1438, 2018.

[26] J. Lee. Issue 1565: Microsoft Edge: Chakra: JIT: ImplicitCallFlags check bypass with Intl. https://bugs.chromium.org/p/project-zero/issues/detail?id=1565, 2018.

[27] C. Lemieux and K. Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485. ACM, 2018.

[28] S. McKenzie. Babel, Javascript compiler, 2019. https://babeljs.io/.

[29] Microsoft. ChakraCore, The core part of the Chakra JavaScript engine that powers Microsoft Edge, 2019. https://github.com/microsoft/ChakraCore.

[30] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. Semantic fuzzing with zest. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Beijing, China, July 2019.

[31] J. Patra and M. Pradel. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664*, 2016.

[32] M. Rajpal, W. Blum, and R. Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[33] M. Rash. A Collection of Vulnerabilities Discovered by the AFL Fuzzer, 2017. https://github.com/mrash/afl-cve.

[34] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[35] S. Sanfilippo. Redis, Open source in-memory database, cache and message broker, 2019. https://redis.io/.

[36] K. Serebryany. libfuzzer a library for coverage-guided fuzz testing. *LLVM project*, 2015.

[37] K. Serebryany. Sanitize, Fuzz, and Harden Your C++ Code. In *Proceedings of the 1st USENIX ENIGMA*, San Francisco, CA, Jan. 2016.

[38] R. Swiecki. Honggfuzz. *Available online a t: http://code. google. com/p/honggfuzz*, 2016.

[39] W. Syndder and M. Shaver. Building and Breaking the Browser. In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2007.

[40] Syzkaller. Syzkaller Found Bugs - Linux Kernel, 2018. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md.

[41] B. Turner. Random c program generator. *Retrieved from*, 2007.

[42] S. Veggalam, S. Rawat, I. Haller, and H. Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *Proceedings of the 21th European Symposium on Research in Computer Security (ESORICS)*, Crete, Greece, Sept. 2016.

[43] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[44] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Montréal, Canada, May 2019.

[45] G. Wen, Y. Zhang, Q. Liu, and D. Yang. Fuzzing the actionscript virtual machine. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Hangzhou, China, May 2013.

[46] D. Yang, Y. Zhang, and Q. Liu. Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1070–1076. IEEE, 2012.

[47] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, June 2011.

[48] H. Yoo and T. Shon. Grammar-based adaptive fuzzing: Evaluation on scada modbus protocol. In *2016 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 557–563. IEEE, 2016.

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

APPENDIX

A. *JavaScript engine bugs found by* DIE

Table VIII lists the bugs discovered by DIE in three known JavaScript engines (*i.e.*, ChakraCore in Microsoft Edge, JavaScriptCore in Apple Webkit and V8 in Google Chrome) with their aspect description.

#	JS Engine	Type	Security	Status	Aspect
1	Ch 1.11.5	Incorrect regular expression parsing		Fixed	Non-ascii regex
2	Ch 1.11.5	Incorrect regular expression parsing		Fixed	Non-ascii regex
3	Ch 1.11.5	Use-after-free due to scope escaping	✓	CVE-2019-0609	A gigantic object literal in a nested function
4	Ch 1.11.7	Incorrect profiling during JIT compilation		Fixed	Infinite recursion with exception handling
5	Ch 1.11.7	Memory corruption in JavascriptArray	△	Fixed	Sorting large arrays w/ custom comparison
6	Ch 1.11.7*	Incorrect profiled state during JIT optimization		Fixed	Circular function references
7	Ch 1.11.7*	Inconsistent behavior between negative NaN and positive NaN		Fixed	Hashing NaN in Set
8	Ch 1.11.7*	Breaking assumption related to the size of inline segment	✓	Fixed	-
9	Ch 1.11.8*	Inconsistency in helper label annotation during JIT compilation		Fixed	`new F()` in `F()`
10	Ch 1.11.9	Writability of read-only property in class		Reported	Overwriting fields of `super`
11	Ch 1.11.9	Writability of constant variable		Reported	Deleting constant fields
12	Ch 1.11.9	Incorrect behavior when overwriting previously deleted variable		Reported	Manipulating fields of built-in objects
13	Ch 1.11.9	Incorrect behavior when calling getter of previously deleted variable		Reported	Manipulating fields of built-in objects
14	Ch 1.11.9	Type confusion between integer and double	△	Reported	JIT - Referencing outer vars in a nested function
15	Ch 1.11.9	Inconsistency between cached value and real value	△	Reported	JIT - Object property binding
16	Ch 1.11.9	Memory corruption while building bytecode	△	Reported	-
17	Ch 1.11.9	Memory corruption while parsing JS code	△	Reported	`eval()` giant statements
18	Ch 1.11.9	Memory corruption in JavascriptArray	△	Reported	`Array.prototype.push()`
19	Ch 1.11.9	Null dereference due to wrong scope analysis	△	Fixed	`with` on outer variables
20	Ch 1.11.9	Breaking assumption related to the size of inline segment	✓	Fixed	JIT - Local variable escape
21	Ch 1.11.9*	Incorrect internal state due to misbehavior of the engine		Fixed†	JIT - Indexing statically declared arrays variably
22	Ch 1.11.9*	Wrong no-return annotation for a function which has return		Reported	Error handling in proxy handlers
23	Ch 1.11.9*	OOB write due to wrong JIT optimization	✓	Fixed	JIT - Defining small compound objects
24	Ch 1.11.9*	Incorrect emitted IR from JIT compilation	✓	CVE-2019-1023	JIT - Inlining small functions
25	Ch 1.11.9*	Use-after-free during JIT	✓	CVE-2019-1300	JIT - Indexing and redefining `TypedArrays`
26	Ch 1.11.9*	OOB read/write due to accessing uninitialized variable during JIT	✓	CVE-2019-0990	JIT - Incorrect induction variable used for array index
27	Ch 1.11.9*	OOB read/write due to wrong JIT optimization	✓	CVE-2019-1092	JIT - Indexing and redefining `TypedArrays`
28	Ch 1.11.9*	OOB read/write due to wrong inlining during JIT	✓	Fixed	JIT - Defining and manipulating small objects
29	JSC 2.24.0	Wrong profiling during JIT optimization	△	WebKit 195991	JIT - Control flow analysis
30	JSC 2.24.1	Invalid indices stored in TypedArrays		WebKit 197353	JIT - Indexing in `TypedArrays`
31	JSC 2.24.1	Type confusion of induction variable during JIT	△	WebKit 197569	JIT - Type speculation
32	JSC 2.24.2	Incorrect assumption while compiling JIT IR		Fixed	JIT - Compiling a built-in function
33	JSC 2.24.2	Incorrect type speculation during JIT		Fixed	JIT - Type speculation
34	JSC 2.24.2	Wrongly yielded exception while handling another exception		Fixed	A gigantic string that causes out-of-memory
35	JSC 2.24.2	Inconsistent behavior of garbage collector from JIT profiling		Fixed	JIT - Compiling a built-in function
36	JSC 2.24.2	Invalid state while handling character		Fixed	JIT - switch case statement
37	JSC 2.24.2	Memory corruption while parsing function	△	Fixed	`eval()` gigantic functions
38	JSC 2.24.2	Memory corruption while handling slow path in JIT code	△	Fixed	Triggering Yarr that JIT compiles regexps
39	JSC 2.25.1	Type confusion due to accessing uninitialized memory region	✓	CVE-2019-8676	JIT - Call context analysis
40	JSC 2.25.1	Use-after-free due to wrong garbage collection	✓	CVE-2019-8673	JIT - Garbage collection
41	JSC 2.25.1	Memory corruption while handling regular expression	✓	CVE-2019-8811	Back reference in regex
42	JSC 2.25.1	Memory corruption while creating regular expression	✓	CVE-2019-8816	Non-ascii regex
43	JSC 2.25.1	Null dereference while accessing HashMap	△	Fixed	A gigantic string that cause out-of-memory
44	JSC 2.25.1	Memory corruption due to race condition in concurrent JIT	△	Fixed	asm.js - Storing an object into a number array
45	V8 8.0.0*	Type confusion between heap and internal object in JIT code	✓	CVE-2019-13730	JIT - Switch case statement
46	V8 8.0.0*	Incorrect loop optimization for JIT IR	✓	CVE-2019-13764	JIT - controllable loop bound
47	V8 8.0.0*	Integer overflow while handling regular expression		Fixed	-
48	V8 8.0.0*	Incorrect redundancy elimination in JIT	✓	CVE-2020-6382	JIT - indexing arrays

Ch: ChakraCore, **JSC**: JavaScriptCore
*Canary version † This bug was reported by us but it is still reachable by DIE due to incomplete fix.
△ The bug is a memory corruption which results in a crash. ✓The bug is confirmed to be exploitable for remote code execution or information leakage.

TABLE VIII: New bugs found by DIE in ChakraCore, JavaScriptCore, and V8. The latest version affected by each bug is specified. In the **Status** column, **Fixed** means the bug was also noticed and patched by developers before we reported the bug.