Fuzzing JavaScript Engines with a Graph-based IR

Haoran Xu
NUDT
Changsha, China
xuhaoran12@nudt.edu.cn

Zhiyuan Jiang*
NUDT
Changsha, China
jzy@nudt.edu.cn

Yongjun Wang*
NUDT
Changsha, China
wangyongjun@nudt.edu.cn

Shuhui Fan
NUDT
Changsha, China
fanshuhui18@nudt.edu.cn

Shenglin Xu
NUDT
Changsha, China
xushenglin@nudt.edu.cn

Peidai Xie
NUDT
Changsha, China
xpd2002@126.com

Shaojing Fu
NUDT
Changsha, China
fushaojing@nudt.edu.cn

Mathias Payer
EPFL
Lausanne, Switzerland
mathias.payer@nebelwelt.net

Abstract

Mutation-based fuzzing effectively discovers defects in JS engines. High-quality mutations are key for the performance of mutation-based fuzzers. The choice of the underlying representation (e.g., a sequence of tokens, an abstract syntax tree, or an intermediate representation) defines the possible mutation space and subsequently influences the design of mutation operators. Current program representations in JS engine fuzzers center around abstract syntax trees and customized bytecode-level intermediate languages. However, existing efforts struggle to generate semantically valid and meaningful mutations, limiting the discovery of defects in JS engines.

Our proposed graph-based intermediate representation, FlowIR, directly represents the JS control flow and data flow as the mutation target. FlowIR is essential for the implementation of powerful semantic mutation. It supports mutation operators at the data flow and control flow level, thereby expanding the granularity of mutation operators. Experimental results show that our method is more effective in discovering new bugs. Our prototype, FuzzFlow, outperforms state-of-the-art fuzzers in generating valid test cases and exploring code coverage. In our evaluation, we detected 37 new defects in thoroughly tested mainstream JS engines.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

fuzzing, JavaScript engine, mutation, control flow, data flow

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690336

ACM Reference Format:
Haoran Xu, Zhiyuan Jiang, Yongjun Wang, Shuhui Fan, Shenglin Xu, Peidai Xie, Shaojing Fu, and Mathias Payer. 2024. Fuzzing JavaScript Engines with a Graph-based IR. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690336

1 Introduction

JavaScript (JS) is the most popular programming language and drives the modern Web [54]. An astonishing 98.8% of websites execute JS on the client side [45]. As of 2022, JS has more code repositories than any other language on GitHub [15]. The JS engine serves as a language processor to compile and execute JS code. It is integrated into Web browsers to facilitate dynamic features of websites. In recent years, a notable series of high-risk vulnerabilities [40] have emerged in widely used JS engines, posing substantial security risks for billions of users. Adversaries can chain successful attacks with an escape from the browser sandbox, gaining unauthorized privileges by crafting malicious Web pages and enticing victims to access them [41]. It is imperative to proactively identify potential defects in JS engines to protect users against attacks.

Fuzzing is an effective automated bug discovery approach for JS engines [21, 24, 35, 44, 46]. Mutation-based fuzzers [53, 56] create new test cases by mutating existing seeds. They are effective in exploring the input space surrounding existing inputs, thereby potentially uncovering unexpected edge cases or vulnerabilities [1, 20, 37, 47, 48].

Efficient mutation operators are key to the performance of mutational fuzzers [19]. Designing effective mutation operators for JS engines raises several challenges: **C1-Validity**. Mutations must produce test cases with correct syntax and semantics to prevent the engine from rejecting the test case during the early stages of the engine's processing (e.g., parsing, semantic analysis). **C2-Semantically meaningful mutation**. The attack surface of JS engines predominantly resides in the engine's backend [42], which process the semantics of JS programs. Syntactic information is largely discarded during the parsing stage. Effective testing of

backend components necessitates semantically meaningful mutations, going beyond mere syntactic changes. Achieving this requires thoughtfully altering the control flow or data flow of the program. **C3-Mutation granularity**. High-quality input corpora such as known proof of concept (PoC) inputs or regression test cases are deliberately designed to deal with vulnerable components like JIT compilers, which have specific control-flow conditions crucial for JIT compilation [37]. Effectively identifying vulnerabilities in JIT compilers necessitates refining the granularity of mutation. This entails preserving the control-flow structure within the seed to retain the trigger conditions, coupled with effective semantic mutations at the data-flow level.

The fuzzer's input representation format dictates possible mutation operators. The representation defines the rules and mechanisms by which mutations are applied to existing seeds. Fuzzers for JS engines fall into three categories based on the representation of the JS program:

- Initially, fuzzers have mutated source code based on byte or token sequences [31, 55]. As this mutation is unaware of the syntax, the resulting test cases are prone to syntax errors, thereby diminishing their validity. The majority of generated inputs fail at the early parsing stage, and cannot proceed to more complex aspects of the implementations.
- Current methods mutate the Abstract Syntax Tree (AST). In contrast to token sequences, mutating the AST facilitates the generation of syntactically correct test cases [37, 47].
- Alternatively, mutation may happen at an intermediate language [20]. Fuzzilli devises a bytecode-level Intermediate Representation (IR) of the JS as the mutation target, aiming to produce high-quality inputs.

However, both AST and bytecode IR, commonly used for mutation, have limitations in exploring vulnerabilities in JS engine backends. AST mutations lack semantic constraints, often yielding test cases with semantic errors that cannot reach the backend (C1). Additionally, mutations on the AST generate many test cases with altered syntax but unchanged semantics, failing to adequately test complex interactions of the backend implementations (C2). Bytecode IR lacks explicit control and data flow, making semantically meaningful mutation implementation difficult (C2). For instance, identifying unused data flow in bytecode IR is challenging, leading to wasteful computing resource consumption due to a substantial number of mutations on invalid data flows. Designing fine-grained mutation operators for comprehensive data-flow-level mutation proves highly challenging, regardless of whether based on AST or bytecode IR (C3). Overall, the inherent characteristics of these representations constrain mutation operator design in fuzzing, rendering neither AST nor bytecode ideal choices for mutation.

We address the aforementioned gap by designing a new representation to support effective mutation operators. The mutation of a program requires code transformation, and one common scenario within this transformation is optimization [27]. Similarly, the mutation operator is closely related to the IR where the mutation is carried out. We introduce a graph IR named FlowIR. Our proposed IR directly represents the JS program's control flow and data flow for mutation. Specifically, we explicitly model the control flow and data flow of JS programs with FlowIR, capturing the semantics of the program through the interconnected relationships between nodes. FlowIR supports bidirectional conversion to and from JS. During fuzzing, we maintain the seed queue in the FlowIR format. Based on FlowIR, we design a series of mutation operators. Mutations are performed on FlowIR, and subsequently, the mutated representation is converted back to source code as input for the tested engines.

To address C1, FlowIR emphasizes representing semantics rather than syntactic structures. This strategy facilitates the enforcement of semantic constraints during mutation. To address C2, FlowIR represents the control flow and data flow of the program directly. Mutations operate directly on the semantics of the seed, simplifying the implementation of meaningful semantic mutations. For instance, explicit modeling of the data flow allows for the easy identification of unused data flows in the seed, enabling the avoidance of mutation on invalid semantics. To address C3, this paper establishes a low coupling between control flow and data flow in FlowIR. This feature enables the fuzzer to concentrate on mutations within either the control-flow or data-flow subgraph independently, mitigating the need to simultaneously address their mutual influence and thereby minimizing the likelihood of introducing semantic errors. This enhances the operational granularity of mutations.

To validate the efficacy of our approach, we implement FuzzFlow as a prototype fuzzer that utilizes FlowIR. The experimental results demonstrate that FuzzFlow enhances the syntax correctness and semantic validity of generated test cases significantly. Low coupling between data flow and control flow enhances testing effectiveness for backend engine components. The validity of test cases generated by FuzzFlow reaches 72% (18.6% higher than the baselines), showcasing a remarkable improvement in code coverage by 4.78%. Moreover, mutations based on the FlowIR achieve high throughput, leading to efficient fuzzing. After applying our technique to six mainstream JS engines (V8 in Chrome, SpiderMonkey in FireFox, JavaScriptCore in Safari, ChakraCore, JerryScript, QuickJS), FuzzFlow has identified a total of 37 new defects. Our prototype will be available at https://github.com/walkcreate/FuzzFlow. We make the following contributions:

- This paper proposes FlowIR, a graph-based program representation used for mutation that directly represents the control and data flow of JS.
- Based on FlowIR, this paper proposes mutation operators with the following advantages: (1) facilitate the generation of semantically valid test cases, (2) enable semantically meaningful mutation, (3) enhance mutation at finer granularity.
- The experimental results demonstrate that FlowIR serves as a highly effective mutation target for fuzzing JS engines. Based on FlowIR, we implemented a new fuzzer named FlowFuzz, which has successfully identified 37 new defects in the mainstream JS engines.

2 Background

2.1 Challenges when Fuzzing JS Engines

Mainstream JS engines are composed of a parser, bytecode compiler, interpreter, JIT compiler, and supporting components. To enhance the execution efficiency, mainstream JS engines adopt a mixed compilation architecture. Specifically, they use a bytecode compiler

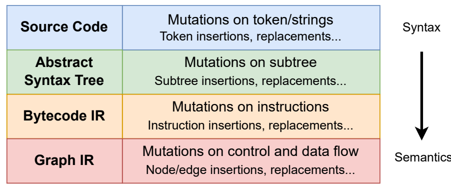


Figure 1: Mutations for JS can be applied at different levels

[18] as a baseline, complemented by one or even multiple levels of JIT compilers [16] to deeply optimize hot code, and compile it into machine instructions for execution. The introduction of a JIT compiler has significantly improved the execution efficiency of JS engines [25]. However, its complexity has also unveiled new attack surfaces, making JIT compilers a focal point for researchers [3, 40].

The attack surface of the JS engine predominantly resides in its backend components, notably represented by the JIT compiler [42] and garbage collector [39]. While the frontend handles lexical analysis and syntax analysis, the backend handles garbage collection, code optimization, and deoptimization. Backend functions involve complex logic with frequent updates. Uncovering defects in backend components imposes elevated demands on the mutation operators employed by fuzzers.

2.2 Existing Mutation Targets and Operators

The JS engine exclusively receives input in the form of source code. Regardless of the chosen IR for mutation, such as AST or bytecode, a conversion process is required between the JS program and the selected IR. Although this conversion introduces computational overhead, opting for an IR instead of the token sequence significantly enhances the quality of generated test cases [47]. This improvement contributes to the overall effectiveness of fuzzing, explaining its widespread adoption among researchers.

The parsing process produces the AST. Given the existence of open-source parsers for most programming languages [2], AST stands out as the most readily accessible IR. In comparison to token sequences, AST-based mutations prove advantageous in preserving the syntactic validity of the code. Therefore, it serves as the predominant mutation target for the fuzzing of language processors [1, 23, 47]. However, mutations based on the AST often cause semantic errors (with the result that test cases are rejected early) [37]. Moreover, as it corresponds directly to the syntax structure, mutating at the AST-abstraction yields alterations in syntax but no change in semantics. Such mutations fail to reach the vulnerable backend of JS engines.

To enhance the capability of mutation operators in generating semantic mutations, Fuzzilli [20] conducts mutations on the bytecode IR developed in their work. They introduce FuzzIL, an IR comprised of instruction sequences. The supported mutation operators encompass the insertion and modification of instructions, as well as the mutation of instruction input. Unlike AST, bytecode-level IR is generated after specific semantic analysis, providing a more direct expression of semantics. However, FuzzIL does not explicitly correspond to the control flow and data flow, potentially resulting in meaningless semantic mutation.

```
1  var v0 = 0;
2  var v1 = "hello";
3  v0 = 3;
4  var v2 = v1 + " world";
5  print(foo(v0, v2));
6  // ------ mutation ------
7  var v0 = 100; // mutation on unused data flow
8  var v1 = "hello";
9  v0 = 3;
10 var v2 = v1 + " world";
11 var v3 = foo(v0, v2); // mutation on syntax only
12 print(v3);
```

Listing 1: An example of mutation on AST

```
1  v0 <- LoadInt 0
2  v1 <- LoadString "test"
3  ...
4  v23 <- LoadInt 0
5  v24 <- LoadBuiltin "foo"
6  CallMethod v24, v0
7  // ------ mutation ------
8  v0 <- LoadInt 0
9  v1 <- LoadString "test"
10 ...
11 v23 <- LoadInt 0
12 v24 <- LoadBuiltin "foo"
13 CallMethod v24, v23 // mutation with the same data flow
```

Listing 2: An example of mutation on Bytecode-level IR

Listing 1 highlights the mutation with changes in syntax but no change in semantics. At the first mutation position, the initial assignment to variable `v0` in the seed is *an unused data flow*, because the value of the variable is reassigned to 3 before `v0` is subsequently read. Therefore, there is no benefit in mutating the value `0` in line 1. Detecting such unused data flows based on the existing mutation targets requires challenging data flow analysis. In the second mutation position, the syntax tree of the test case has changed, yet the semantics of the program remain unchanged. Unfortunately, these mutations are inevitable with existing fuzzers. This type of mutation is incapable of testing the backend components of the JS engine.

To enhance the mutation of the seed's data flow, Fuzzilli introduces several constraints on the bytecode IR. These constraints involve adopting a Static Single Assignment (SSA) paradigm [8, 9] and restricting instructions to accept only a single variable as input. While these constraints enhance the efficacy of semantic mutation, they do not fully address the underlying challenge. For instance, Listing 2 illustrates the analytical dilemma encountered when implementing semantic mutation based on bytecode IR. Variable `v0` and `v23` differ, yet they contain identical data. As a result, changing `v0` to `v23` does not modify the seed's semantics, presenting a challenge in identifying these indistinguishable data flows based on bytecode IR.

2.3 Program Transformation and Graph IRs

The mutation of JS seeds involves transforming these programs. Compilation optimization transforms code to be more efficient.

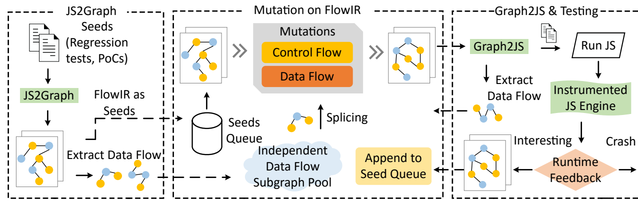


Figure 2: Overall architecture of FuzzFlow

However, a notable distinction is that optimization requires preserving the semantic invariance of the program, a constraint not applicable to mutation.

Optimizations are accomplished on IRs. IR serves as an intermediate layer between the source code and the machine code [28, 43]. Modern compilers often employ a variety of IRs. Common examples include the AST, bytecode IR, or graph-based IR. The AST represents syntax structure but lacks a direct representation of program semantics, resulting in challenges for the execution of semantic-related optimizations. Bytecode IR involves complex expression of control flow and data flow [7]. Optimizing at this level may require dealing with intricate branching and data manipulation, making it more challenging compared to higher-level representations.

Graph-based IRs [4] play a crucial role in modern compiler design by providing a structured unified representation that facilitates a wide range of analyses and optimizations across different programming languages [6]. Graph IRs utilize graph-based data structures to depict the control flow and data flow of a program. Generally, nodes symbolize program entities (e.g., constants, expressions, statements), and edges between nodes signify relationships or dependencies among program entities. Presently, graph-based IRs are widely used in modern compilers and contribute to a range of optimization tasks, including constant propagation, common sub-expression elimination, loop optimization, or function inlining [13]. Program Dependency Graph (PDG) [14] stands out as a widely adopted paradigm within the realm of graph IR. PDG explicitly models and represents the control-flow dependencies and data-flow dependencies of the program, offering a structured representation that assists the efficient deployment of many traditional optimization techniques. Illustrating on the JS engine, the TurboFan compiler [16] employed by V8 is developed on the graph-based TurboFan IR. Edges in TurborFan IR represent data flow, control flow, and dependencies.

Inspired by the code transformation scenario of optimization, this paper advocates for the use of graph IR as a mutation target to effectively achieve semantic mutations of JS programs. This is depicted in Figure 1.

3 Design

Figure 2 highlights the overall framework of FuzzFlow, the pioneer fuzzer that mutates programs using a graph IR. FuzzFlow comprises three components: *JS2Graph*, *Mutation*, and *Graph2JS*. In the fuzzing initiation, *JS2Graph* compiles the initial seed set into FlowIR format. While fuzzing, *Mutation* randomly picks a seed from the queue and mutates it. Post-mutation, *Graph2JS* lifts test cases in FlowIR back to source code for execution. We detail the modules as follows.

3.1 FlowIR

Our main contribution, FlowIR, expressively represents the control flow, data flow, and dependencies between JS entities. Our prototype implementation, FuzzFlow, utilizes FlowIR for JS fuzzing.

The first key difference between FlowIR and existing graph IRs is that FlowIR supports bidirectional conversion with source code, achieved through a careful redesign of nodes and edges. Bidirectional conversion enables the fuzzer to reflect on and improve seeds continuously, mapping execution feedback to concrete parts of the test input.

The second key difference between FlowIR and existing IRs is the precise definition of FlowIR's functional scope, which prevents exposure to unnecessary information during mutation. This feature allows the fuzzer to avoid redundancy in the test inputs. Nodes are categorized into two types: Control Flow Nodes (CFN) capturing the program's structural representation, and Data Flow Nodes (DFN) handling data-related operations. The label on a node indicates the operation it represents, with inputs to a node serving as inputs to the operation.

To construct FlowIR from a JS program, we conduct an intra-procedural analysis, associating each method with a corresponding graph. We merge control flow and data flow into a cohesive graph with minimal coupling. This unified graph provides two key benefits: it aids in JS code conversion and facilitates mutation operators involving both control flow and data flow, thereby enhancing flexibility in mutation operator design. The coupling is low, allowing independent mutations of control flow and data flow.

Definition 1: The FlowIR $G = (V, E, \lambda)$ is a directed, node-labeled, and edge-unlabeled graph where V is a set of nodes, $E \subseteq (V \times V)$ is a set of directed edges, and $\lambda\colon N \longrightarrow \Sigma$ is a node labeling function assigning a label from the alphabet Σ to each node.

This graph encompasses a Control-Flow subGraph (*CFG*) and a Data flow Dependency subGraph (*DDG*):

- Nodes represent control structures in the program, as well as operators and operands. The labels on the nodes represent the node's semantic operation.
- The edges incident to a node represent both the data values on which the node's operations depend and the control conditions on which the execution of the operations depends.

The set of all dependencies for a program are viewed as inducing a partial order on the operations in the program that must be followed to preserve the semantics of the original program. For improved analysis, FlowIR is in SSA form. Each node produces at most one value. ϕ functions are used at join points. ϕ nodes input several data values and output a single selected data value.

Definition 2: The *CFG* represents the partial order of statement execution, as defined by the semantics of JS.

The *CFG* comprises two node types: operator or instruction nodes with side effects (behavioral CFNs), and region nodes (structural CFNs) indicating control flow structures. Operator nodes with side effects include *New* and *Delete* for object creation and deletion, *Load* and *Store* for reading and writing objects, and *Invoke* for method calls. Instruction nodes denote control flow changes like return and throw statements. Operator and instruction nodes in *CFG* are fixed within the control flow. Motion of these nodes

will change the semantics of the program. Region nodes encapsulate control conditions, grouping nodes with identical conditions. FlowIR employs region nodes to represent control structures such as branches, loops, and exceptions. The distinguished entry node *Start* representing the beginning of the program.

Nodes in the *CFG* have *successor* edges indicating their possible next nodes. Edges between control flow nodes denote direct transitions. Each node in the graph has at most two successors. Nodes with two successors are assumed to have true and false attributes associated with their outgoing edges.

Definition 3: The *DDG* demonstrates the flow of values from definitions of a variable to its uses. The nodes in *DDG* consist of operators and operands.

For data flow representation, a node has *input* edges pointing to nodes providing its operands. The inputs to a node are inputs to the node's operation. Operands encompass literals, variables, or expressions. Each node defines a value based on its inputs and operation, available on all output edges. All *input* edges signify scheduling dependencies. A node must be scheduled post its dependencies when lifting FlowIR to JS.

Control-flow structures form a backbone for data-flow nodes. Data flow nodes are solely restricted by their data dependencies. Formally, let X and Y be nodes in a *DDG*. There is a data dependence from X to Y with respect to a variable v iff:

(1) there is a non-null path p from X to Y with no intervening definition of v and either:
(2) X contains a definition of v and Y a use of v; or X contains the use of v and Y a definition of v.

Edges between CFG and DDG: FlowIR minimizes the coupling between control flow and data flow. Interaction between the CFG and DDG is restricted to three node types: behavioral CFNs, *PhiNode* and *IfNode*. Behavioral CFNs often take data flow input. Meanwhile, they may have control predecessors and successors. *PhiNode* is classified into data flow nodes, linked to control condition expressions and resulting data-flow nodes from two branches. The ith data input to *PhiNode* corresponds to the ith branch. Similarly, *IfNode* is classified into a control flow node, linked to both the data flow node representing the conditional expression and the two control flow branches. We show the example edges between CFG and DDG in Figure 3.

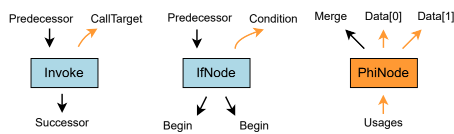


Figure 3: Nodes connecting control flow and data flow

In our prototype, FlowIR incorporates support for diverse JS language features, covering basics like variable operations, binary/ternary operations, if-else branches, loops, and functions. Additionally, FlowIR extends support to advanced features, including JS object-oriented programming, and the JS exception-handling mechanism. Nodes for fundamental language features are language-independent, while those expressing JS-specific semantics are not. This implies that FlowIR cannot be directly used as a mutation target

Algorithm 1: Translate JS to FlowIR

```
   Input  :The seed JS code
   Output:FlowIR of the seed
1  ast ← Parse(JSCode)
2  ast ← ScopeAnalysis(ast)
3  ast ← ReferenceResolve(ast)
4  ast ← LeftValueAnalysis(ast)
5  graph ← CreateEmptyGraph ()
6  function ProcessTree(node):
7      if node == VariableDeclaration then
8          if node.initialization is not null then
9              flowNode ← ProcessTree (node.initialization)
10             Create a new variable proxy
11             Set the variable proxy to flowNode
12             AddNodeToGraph ()
13             return flowNode
14     if node == BinaryOpExpression then
15         leftFlow ← ProcessTree (node.left)
16         rightFlow ← ProcessTree (node.right)
17         Create BinaryOpNode
18         Create edges between BinaryOpNode and its dependencies
19         AddNodeToGraph ()
20         return BinaryOpNode
21     if node == IfStatement then
22         conditionFlow ← ProcessTree (node.condition)
23         Create BeginNodes and EndNodes for the two branches
24         Create IfNode
25         Create edges between IfNode and its dependencies
26         Set the current control flow as the BeginNode for true branch
27         ProcessTree (node.trueBranch)
28         Set the current control flow as the BeginNode for false branch
29         ProcessTree (node.falseBranch)
30         Create MergeNode
31         Merge the two branches control flow into MergeNode
32         AddNodeToGraph ()
33         Set the current control flow as the MergeNode
34         return IfNode
35     ...
36     return node
37 ProcessTree (ast)
```

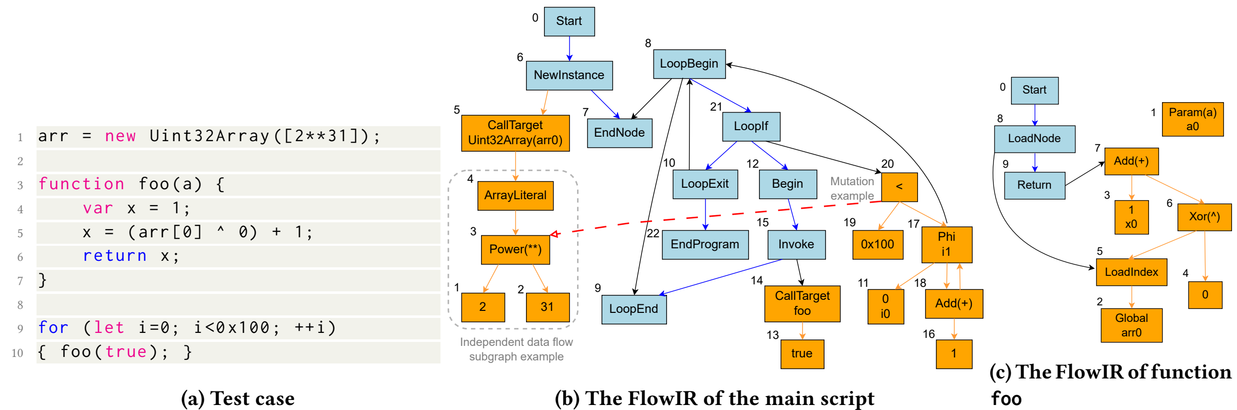
for other language processors. However, the conceptual foundation of FlowIR can be extendable to other languages. Moreover, nodes for fundamental language features are reusable.

Figure 4 illustrates the FlowIR corresponding to a seed triggering CVE-2021-21220 in V8. Notably, FlowIR directly represents the control flow and dependencies in data flow, achieving low coupling. Data flow analysis on FlowIR is straightforward. For example, in Figure 4c, node-1 is unused by any other, indicating an unused data flow (parameter in function `foo`). Consequently, mutating this node is ineffective for uncovering defects in the JS engine.

Mutations on FlowIR are highly efficient. For instance, altering the edge from node-20 to node-19 in Figure 4b and redirecting it to node-3 (red dashed line) allows mutation of the loop condition from `i < 0x100` to `i < 2**31`. Notably, the latter expression corresponds to an AST subtree with a height of 2. Generating such an AST subtree requires 2 steps. In contrast, leveraging the graph, only modifying the destination node of an edge is needed.

3.2 Translating JS to FlowIR

To leverage existing high-quality seeds, FuzzFlow supports the translation of JS to FlowIR. Notably, Lee et al. [29] underscore the effectiveness of regression test cases as fuzzing seeds. Development teams for mainstream JS engines continually augment their regression test sets with tests that expose historical engine defects, aiding



(a) Test case

(b) The FlowIR of the main script

(c) The FlowIR of function `foo`

Figure 4: Test case to trigger CVE-2021-21220 in V8 and the corresponding FlowIR. Blue nodes denote control flow nodes, while orange nodes signify data flow nodes. Blue edges represent control flow edges, while orange edges represent data flow edges. Black edges denote connections between control flow and data flow, as well as auxiliary connections.

subsequent development. By mutating these regression test cases, the fuzzer gains an avenue to delve deeper into historical vulnerability patterns. To allow continuous testing and protect against regression bugs, we implement the *JS2Graph* module, designed to seamlessly convert JS programs into FlowIR. Consequently, FuzzFlow can navigate the input space surrounding existing test cases, thereby revealing potentially unforeseen neighboring bugs.

The *JS2Graph* involves syntax analysis of the JS to acquire the AST. Subsequently, a top-down semantic analysis is conducted on the AST to establish scope, identify declared symbols, perform resolve resolution, and analyze left values. Finally, syntax-directed translation is performed based on the results of the semantic analysis to convert the AST into FlowIR. Throughout this conversion, the control flow and data flow of the program are analyzed.

Algorithm 1 outlines the conversion process of JS2Graph. Given the diverse syntax features of JS and space constraints on the page, the algorithm only presents a subset of JS2Graph functionality. Considering that the basic building blocks of JS programs are declarations (e.g., VariableDeclaration), expressions (e.g., BinaryOp, represented by the DFN), and statements (e.g., IfStatement, represented by the CFN), Algorithm 1 opts to showcase these three elements. The prototype implementation encompasses support for additional language features.

3.3 Mutation Operators on FlowIR

FuzzFlow encompasses two types of mutation operators: data-flow subgraph mutation and control-flow subgraph mutation.

Mutation on Data-flow Subgraph. The data-flow subgraph mutation entails preserving the control structures of the seed while solely mutating the data flow. This mutation operator introduces a finer granularity. For instance, it enables the retention of conditions that trigger JIT compilation. If the seed induces JIT compilation, the mutated test cases under this operator can also similarly trigger JIT compilation.

Data flow mutation via FlowIR provides distinct advantages over Fuzzilli's FuzzIL, a sequential IR based on Three Address Code (TAC). Firstly, TAC's data flow is fixed within the control flow structure of the JS program. However, this mutation is restricted by instruction sequence, limiting the available data flow to that generated by preceding instructions. Consequently, the mutation space is constrained, hindering full utilization of the seed's overall data flow. Secondly, input mutation of TAC instructions alters variable names as instruction parameters. As noted in the background, variable names do not directly map to the data flow. Different variable names may represent the same data flow, posing challenges for effective data flow mutation. FlowIR effectively addresses the aforementioned challenges. Firstly, FlowIR's DFNs are independent of control flow, constrained only by data dependencies. After changing these dependencies through mutation, the *Graph2JS* module can appropriately relocate DFNs within control flow regions. Thus, leveraging FlowIR's data-flow subgraph enables mutations to span instructions and basic blocks, facilitating comprehensive analysis and mutation of the entire seed's data flow. Secondly, the input for mutation is the DFN, not the variable name. DFNs in FlowIR accurately model data dependencies within the seed, enabling more targeted mutations.

Data-flow subgraph mutation operators can be further categorized into two types. The first category involves **node attribute mutation**. This fine-grained mutation alters the attributes (labels) of DFNs. For example, if an original binary operation node employs an addition operator, mutating it into a subtraction modifies the data flow of the seed. Similarly, for a terminal node with an original integer value of 1, changing its value to −1 alters the data flow of the seed. The second category is **input mutation** of the node, which is further divided into intra-procedural mutation and inter-procedural mutation. In intra-process mutation, the data flow is mutated by modifying the connection between data nodes and changing the input of a specific data node within the method. Leveraging FlowIR, this mutation demonstrates high execution efficiency. It involves changing the starting point of an input edge of the data node to another one without having to copy any nodes. Inter-procedural mutation on data flow aims to splice the flow across methods. **Independent data-flow subgraphs** are extracted from the seeds to serve as the material for splicing.

Figure 4b shows an independent data-flow subgraph in the gray rounded rectangle. An independent data-flow subgraph is defined as a subgraph extracted from the FlowIR. In this context, all nodes within the subgraph depend solely on the data flow inputs existing in that subgraph. The extraction of independent data-flow subgraphs aims to establish a comprehensive data flow *Pool* for mutations. During initialization of a fuzzing run, after converting all seed JS programs to FlowIR, we analyze each FlowIR and extract independent data-flow subgraphs into the *Pool*. Each node within any subgraph in *Pool* has the potential to be inserted into a seed as an input node for splicing, thus achieving data flow splicing between procedures.

To extract independent data-flow subgraphs, the traversal initiates from the leaf nodes, typically literals, within the data-flow subgraph, ascending along their dependencies. Leaf nodes, without data input, are unequivocally eligible for inclusion in the data-flow subgraph. Following this, the analysis advances along the data flow edge of each node, examining the higher-level nodes that utilize the current node as their data input. For each traversed DFN, whether it is included in the subgraph depends on all the DFNs it depends on already being part of the subgraph. If an input (designated as I) on which the DFN relies is absent from the extracted subgraph, there is an attempt to recursively incorporate node I into the subgraph. When executing input mutations of data flow across graphs, an independent subgraph is randomly selected from *Pool* as the input for the mutation node.

Mutation on Control-flow Subgraph. In comparison to existing mutation platforms, control flow mutation based on FlowIR proves advantageous in generating test cases that are both syntactically correct and semantically valid. When mutating the control flow in AST or bytecode IR, changes in the positioning of control flow elements can easily break the semantic integrity of the seed. Conversely, FlowIR, by modeling dependencies between nodes, ensures that moving a CFN does not impact other edges of the node, except for the mutated control flow edge. The reason is that dependencies related to unchanged edges are automatically fulfilled in the *Graph2JS* module. This helps control flow mutation based on FlowIR without compromising semantic dependencies. Such a mechanism contributes to the generation of test cases that are not only syntactically correct but also semantically valid.

The mutation target of the control-flow subgraph may be a single behavioral CFN, or a node group containing a control flow structure, such as a branch or a loop. Control flow mutation operators can be further divided into four types: *CFN scheduling*, *CFN deletion*, *CFN insertion* and *CFN data input mutation. CFN scheduling* involves alterations in the placement of CFNs, which can markedly modify the execution of the seed and induce semantic changes. Specific implementation methods for control flow scheduling include:

- Individual CFN Movement: This mutation relocates a behavioral CFN (e.g., function call or object creation) to a random position in the control flow.
- CFN Group Relocation: This method entails moving a control flow unit, such as conditional branches or loops, to a random location in the control flow.
- If-Else Branch Swapping: This mutation involves swapping the if-else branches, thereby changing the execution branch under specific branch conditions.

Control flow scheduling proves highly effective in altering the execution of a seed. For instance, it can shift the control flow from within a loop to outside the loop, or vice versa. Leveraging the graph data structure, *CFN deletion* requires only the modification of predecessor and successor edges between CFNs to achieve the desired changes. *CFN insertion* involves incorporating new nodes or node groups into the existing flow. For instance, in a seed containing function declarations, one can introduce nodes for new function calls with different parameters. Additionally, loops can be inserted into a seed, thereby augmenting the complexity of its control structure. *CFN data input mutation*, coupling control flow, and data flow, specifically modifies the data input of CFNs. For instance, altering the data input of a `ReturnNode` impacts the data flow that is returned. In most cases, when mutating the control-flow subgraph, changes in the data-flow subgraph are often implicated, indicating a larger mutation granularity.

Speeding Up Mutators. We conduct further optimizations on FlowIR-based mutations to enhance the fuzzer's efficiency. Instead of copying the entire graph for a mutation, we conduct the mutation directly on the FlowIR of the seed, keeping a record of the performed mutation operators. After mutation, the *Graph2JS* translation is applied to the mutated FlowIR to generate the JS program for testing. When the engine finishes execution, we copy the mutated FlowIR to generate a new seed only when the test case reveals new coverage. Irrespective of the interest of the mutated seed, after the execution finishes, a reverse operation is performed according to the recorded mutation operators, restoring the seed to its pre-mutated state. This design mitigates unnecessary copying of FlowIR, thereby improving the overall efficiency of the fuzzer.

3.4 Translating FlowIR to JS

After mutation, FuzzFlow converts FlowIR back into JS as fuzzing input. This conversion poses new challenges. Firstly, the IR must preserve high-level semantic information to facilitate the conversion back to JS programs, distinguishing it from existing IRs. For example, in LLVM IR, the control flow of exception handling is simplified to ordinary branch and jump instructions, resulting in the loss of crucial high-level semantic information that makes it impractical to restore the IR to source code. Secondly, the conversion must ensure the accurate translation of both control and data flow, maintaining consistency between FlowIR and the source code.

To address the aforementioned challenges, FuzzFlow incorporates targeted designs. Firstly, in FlowIR, high-level semantics are preserved as nodes. By utilizing these nodes (e.g., TryNode, StoreFieldNode), precise restoration of semantics is achieved. Secondly, FuzzFlow conducts graph traversal along the control flow based on the dependencies and completes code generation during this traversal process. Dependencies are satisfied during this process. When lifting the FlowIR, FuzzFlow structures the source code in *regions.* A *region* represents a nested structure, corresponding to a block scope of the program. Organizing the generated code in *regions* helps maintain the relationships between scopes.

Algorithm 2: Translate FlowIR to JS

```
   Input  :FlowIR of the seed
   Output:JS code of the seed
1  StartNode ⟵ FlowIR
2  topRegion ⟵ CreateRegion ()
3  function LiftGraphNode(node):
4      if node == BinaryOpNode then
5          region ⟵ LoadCurrentRegion ()
6          proxies ⟵ GetVariableProxyOnDataNode ()
7          if IsFirstVisit then
8              leftCode ⟵ LiftGraphNode (node.left)
9              rightCode ⟵ LiftGraphNode (node.right)
10             binaryCode ⟵ leftCode + node.operator + rightCode
11         if len(proxies) > 0 then
12             GenAssignmentStatement (region)
13             return proxies.name
14         else
15             return binaryCode
16     if node == IfNode then
17         parentRegion ⟵ SaveCurrentRegion ()
18         ifRegion ⟵ CreateRegion ()
19         conditionCode ⟵ LiftGraphNode (node.condition)
20         EmitExpression (ifRegion, conditionCode)
21         branchTrueRegion ⟵ CreateRegion ()
22         Change current region to branchTrueRegion
23         LiftGraphNode (node.branchTrue)
24         branchFalseRegion ⟵ CreateRegion ()
25         Change current region to branchFalseRegion
26         LiftGraphNode (node.branchFalse)
27         Add two sub-regions to ifRegion
28         Restore the parentRegion as current region
29         LiftGraphNode (node.next)
30         return
31     ...
32     return
33 LiftGraphNode (graph)
34 JSCode ⟵ MergeRegionsCode (topRegion)
```

Algorithm 2 illustrates the lifting process of certain FlowIR nodes. The prototype FuzzFlow encompasses support for the full list of FlowIR nodes. The *Graph2JS* initiates from the control flow starting node and proceeds backward along the flow. When encountering CFNs that generate scopes, such as `IfBlock`, `ForLoop`, or `WhileLoop`, a new *region* is created to accumulate the code that needs to be generated within the scope. When a CFN depends on any DFN, a depth-first traversal of the data flow is performed along the dependencies. If variables are associated with the DFN, variable assignments are generated. After traversing the current CFN and the dependent DFNs, the traversal continues backward along the control-flow dependencies to subsequent nodes. Once the traversal of the entire graph is completed, the statement sequence within the nested *region* is consolidated into a complete program and passed to the test module.

3.5 Implementation

To demonstrate the effectiveness of our proposed method, we implement a fuzzer (FuzzFlow) that leverages FlowIR. Given the pivotal role of mutation target and operators in fuzzer implementation, crafting FuzzFlow based on existing open-source fuzzers would require us to change over 90% of the code. Therefore, we implement FuzzFlow from scratch with C++. FuzzFlow is a coverage-guided grey-box fuzzer that we implement in 19K lines of code (LoC). We use clang sanitizer coverage as feedback to guide the fuzzer to explore the code coverage of the JS engine. The *JS2Graph* module first performs syntax analysis on the input JS. We use ANTLR [38] to implement the parsing component, and implement the rest of *JS2Graph* ourselves. Fuzzer components including *Graph2JS* and *Mutation* are implemented entirely in C++.

We rigorously tested the *JS2Graph* and *Graph2JS* modules, ensuring the accuracy of the JS-to-Graph and Graph-to-JS conversion processes with numerous unit tests. *JS2Graph* only operates during the fuzzing startup phase to handle the initial seed set, encountering high-quality JS test cases, including historical PoCs. Therefore, the primary challenge lies in accurately translating JS language features. Unit tests help confirm the accurate translation of supported JS language features.

Conversely, *Graph2JS* encounters issues with malformed FlowIR due to mutation, resulting in some seeds being unable to convert back to JS. After running FuzzFlow for 24 hours, we found that, in the current version, 8% of mutated Graph instances cannot be lifted to JS. We can address this challenge by applying semantic constraints to FuzzFlow's mutation operators, which will help mitigate the issue. This optimization will be a focus in future development iterations. It is worting noting that the possible failure of converting mutated IR back to JS is not exclusive to FuzzFlow. When mutations are applied based on IRs like AST, fuzzers encounter similar issues, as mutations can disrupt the original well-formed structure, rendering conversion back to the source code impossible.

4 Evaluation

Evaluation Targets. We evaluated a total of six mainstream JS engines. The targeted engines encompass Chrome V8, Firefox SpiderMonkey, Safari JavaScriptCore, and ChakraCore designed for desktop browsers. Additionally, we tested JerryScript and QuickJS, which are often deployed on IoT devices. There are currently no benchmarks, such as LAVA-M [11] or Magma [22], specifically designed for evaluating JS engine fuzzers. Meanwhile, the number of mature industrial-grade JS engines is limited. Therefore, despite ChakraCore not being utilized in the Edge browser, given its status as an industrial-grade JS engine developed by Microsoft over several years, we consider it an interesting target.

These JS engines have undergone thorough code audits and testing conducted by both the development team and security researchers. Any newly detected defects by FuzzFlow have been missed by earlier evaluations, affirming the efficacy of the method proposed in this paper.

Experimental Setup. Both V8 and SpiderMonkey already include built-in support for Fuzzilli. To ensure compatibility, we adhere to the Fuzzilli interface. For the other four engines, we modified the engine code slightly based on Fuzzilli's patch method. This modification enables communication between the fuzzer and the JS engine through pipelines for test case delivery and coverage feedback. To prevent the fuzzer from becoming stuck on non-terminating processes, the timeout mechanism is commonly employed by fuzzers. The timeout interval represents the duration permitted by the fuzzer for executing a single test case. In our experiments, we adopted the default timeout of Fuzzilli, which is set to 250ms.

To enhance bug detection, JS engine developers have incorporated numerous assertions into the engine source code for internal checks. Assertion errors within JS engines may indicate significant

Table 1: New bugs found by FuzzFlow. The two bugs in V8 are also noticed and patched by developers before our reporting.

#	JS Engine	Issue ID	Component	Security	Status	Description
1	V8	-	Torque		Fixed	Fatal error in string-tq-inl.inc
2	V8	-	JIT compiler	✓	Fixed	Fatal error in deoptimizer.cc
3	JavaScriptCore	261949	Runtime			Assertion error in runtime/SparseArrayValueMap.cpp
4	JavaScriptCore	265272	JIT compiler	✓	Fixed	Integer calculation error after JIT optimization
5	SpiderMonkey	1849099	JIT compiler		Fixed	Incomplete patch for bug-1745907
6	SpiderMonkey	1849100	Debugger		Duplicate	makeDebuggeeNativeFunction allows to create a copy of native function
7	SpiderMonkey	1851135	Debugger		Fixed	Incomplete patch for bug-1845270
8	SpiderMonkey	1852729 (CVE-2023-5728)	Garbage collector	✓	Fixed	weakRefMap is updated when a WeakRef target is cleared
9	SpiderMonkey	1853488	JIT Compiler		Duplicate	FoldConstants optimization reduces the conditional expression
10	SpiderMonkey	1863183	Runtime			weakRefMap contains a dead wrapper
11	SpiderMonkey	1864246	Builtins		Fixed	Incorrect conditional unwrapping. Regression from bug 1841118
12	SpiderMonkey	1864257	Builtins		Fixed	Regression from bug 1848467
13	ChakraCore	6944	JIT compiler	✓		Assertion error in Backend/FlowGraph.cpp
14	ChakraCore	6945	JIT compiler	✓		Assertion error in Backend/GlobOptArrays.cpp
15	ChakraCore	6946	JIT compiler	✓		Assertion error in Backend/LinearScan.cpp
16	ChakraCore	6947	JIT compiler	✓		Assertion error in Backend/FlowGraph.cpp
17	ChakraCore	6948	Runtime			Assertion error in Runtime/Language/ValueType.cpp
18	ChakraCore	6949	JIT compiler	✓		Assertion error in Backend/FlowGraph.cpp
19	ChakraCore	6951	JIT compiler	✓		Assertion error in Backend/BackwardPass.cpp
20	ChakraCore	6959	JIT compiler	✓		Assertion error in Backend/TempTracker.cpp
21	ChakraCore	6960	JIT compiler	✓	Confirmed	Assertion error in Backend/BailOut.cpp
22	ChakraCore	6961	Runtime			Assertion error in Library/ScriptFunction.h
23	ChakraCore	6962	Runtime			Assertion error in Types/RecyclableObject.h
24	ChakraCore	6963	Runtime	✓	Confirmed	Segmentation fault
25	ChakraCore	6964	Runtime	✓	Confirmed	Assertion error in Runtime/Language/JavascriptConversion.cpp
26	ChakraCore	6965	Code generator		Confirmed	Segmentation fault
27	QuickJS	192	Bytecode emitter		Fixed	Null Pointer Dereference
28	QuickJS	198	Garbage collector	✓	Fixed	Heap use after free
29	JerryScript	5097	ByteCode generator	✓		Null pointer dereference
30	JerryScript	5098	ByteCode generator			Assertion error
31	JerryScript	5099	JIT compiler			Assertion error
32	JerryScript	5100	JIT compiler			Assertion error
33	JerryScript	5104	Frontend			Segmentation fault
34	JerryScript	5105	Frontend			Segmentation fault
35	JerryScript	5117	Frontend		Fixed	Segmentation fault
36	JerryScript	5118	Frontend			Assertion error
37	JerryScript	5119	Frontend		Fixed	Assertion error

security vulnerabilities. The identification of many high-risk vulnerabilities, such as CVE-2019-8622, often originates from assertion errors. Therefore, similar to Fuzzilli, we utilized the debug configuration when compiling the engine to capture assertion errors.

Additionally, for the triggering conditions of JIT compilation, we have employed the same processing method as Fuzzilli, specifically by lowering the threshold of repeated executions needed to trigger JIT compilation. Typically, we configured the thresholds so that approximately 100 executions trigger the compilation of a function. This threshold strikes a balance, allowing ample iterations for the engine to gather type information while speeding up the fuzzing.

Initial seeds and Experiment platform. We selected test cases, independent of external harnesses, from the regression test suites of the six engines to create the initial seed set. The combined seed set is for all engines. These regression test suites are readily accessible in the engine's repository. In total, we gathered 1,280 test cases from regression test suites. When performing comparative experiments, this seed set served as the initial seeds for baselines. We conducted experiments for RQ1, RQ3, RQ4, and RQ5 on an Ubuntu 20.04 LTS system with an Intel Xeon Gold 6238R (56 cores) and 128 GB RAM, using an RTX 3090 for neural network baselines. RQ2 was evaluated on an Ubuntu 20.04 system with two AMD EPYC 9654 processors (192 cores) and 512 GB RAM.

Baselines. We conducted a comparative analysis of FuzzFlow against four state-of-the-art mutation-based JS engine fuzzers: Superion [47], DIE [37], Montage [29], and Fuzzilli [20]. FuzzFlow and four baselines are all general-purpose JS engine fuzzers, not targeting specific components [10, 48]. Superion, DIE, and Montage mutate the AST of the seed, while Fuzzilli mutates the bytecode IR. These four competitors represent the two currently prevalent types of mutation targets.

In addition to the mentioned baselines, open-source JS engine fuzzers also include CodeAlchemist [21] and jsfunfuzz [35]. However, these two fuzzers are both generation-based and differ from the mutation target explored in this paper. Furthermore, in previous evaluations [29, 37], Montage demonstrated superior performance compared to CodeAlchemist and jsfunfuzz, while DIE outperformed CodeAlchemist. Thus, we selected these four advanced mutation-based fuzzers as comparison targets.

As the paper concentrates on mutation targets and operators, the evaluation aims to scrutinize their impact on fuzzing. Superion, DIE, Montage and FuzzFlow are mutation-based fuzzers. In contrast, Fuzzilli can create new test cases through two methods: generation and mutation. Initial versions of Fuzzilli lacked a JS-to-bytecode compiler, using a generative engine to prepare the seed set. This allowed it to run without initial seeds. Currently, Fuzzilli includes a compiler module for JS-to-bytecode conversion. We provided Fuzzilli with the same seed set as other fuzzers while deactivating its generative engine for a fair evaluation.

Evaluation design. We aim to answer the following research questions through our experiments:

RQ1: Can FuzzFlow find new bugs in real-world JS engines?

RQ2: How does FuzzFlow perform in terms of code coverage and bug finding against state-of-the-art fuzzers?

RQ3: Does FuzzFlow's improvement in mutation granularity help trigger the vulnerable components of the engine?

RQ4: Does FuzzFlow generate correct JS code, both syntactically and semantically?
RQ5: Does using FlowIR as the mutation target introduce significant runtime overhead?

To measure and compare code coverage, bug finding, throughput, and correctness, we conduct 10 rounds of experiments. We then test the statistical significance of FuzzFlow achieving better performance than baselines using Vargha Delaney $\hat{A}_{12}$ and Mann Whitney U test (U). U tests whether a list of observations is stochastically greater than the other list, while $\hat{A}_{12}$ measures the magnitude of the difference (effect size). Finally, we analyze the bug-triggering test cases generated by FuzzFlow through case studies to further discuss the effectiveness of our method.

4.1 RQ1: Identified Bugs

We first evaluate FuzzFlow's bug-finding capability on real-world engines and demonstrate the defects detected by FuzzFlow. To evaluate FuzzFlow's ability to find unknown defects, we use FuzzFlow to conduct testing on six JS engines for 120 days. We allocate 15 cores to each engine. FuzzFlow has detected a total of 37 new defects, including 2 in V8, 2 in JSC, 8 in SpiderMonkey, 14 in ChakraCore, 9 in JerryScript, and 2 in QuickJS. Developers have confirmed 16 bugs, two bugs were duplicates. 12 have been fixed. All confirmed bugs in SpiderMonkey and QuickJS have been fixed. Two bugs in V8 were concurrently patched.

Table 1 shows details of all detected bugs. Many of the identified defects originate from assertion errors. Some of these assertion errors can be classified as security-related based on the bug locations and crash messages (e.g., Issue-198 in QuickJS). However, a comprehensive analysis is required for the remaining assertion errors to ascertain their exploitability, which is a task beyond the scope of this paper. We have submitted all bug information to the developer teams for thorough examination.

Note that FuzzFlow discovered bugs in various components of the JS engine. For instance, we have detected defects in the bytecode emitter, debugger, and runtime of SpiderMonkey. Besides, FuzzFlow also detects deep bugs in the JIT compilers. We have detected 11 bugs in the JIT compilers of 4 browser engines. Among them, 8 out of 14 defects detected in the ChakraCore are located in the JIT compiler. This result demonstrates the advantages of FlowIR as a mutation target in finding deep vulnerabilities.

4.2 RQ2: Bug-finding Ability and Exploring Code Coverage

General-purpose fuzzers (e.g., AFL and its variants) are evaluated through ground-truth benchmarks like Magma or FuzzBench [34]. However, no such benchmark exists for JS engines, necessitating validation on real software without ground truth. To evaluate the bug-finding capabilities of various fuzzers, we tested FuzzFlow and baselines for 360 hours each on V8, SpiderMonkey, JavaScriptCore, ChakraCore, JerryScript, and QuickJS. Ten instances were run per fuzzer-target pair to reduce randomness. After deduplication based on crashing stacks, FuzzFlow, Superion, DIE, Montage and Fuzzilli detected an average of 5.7, 3.5, 2.6, 2.8, and 4.9 defects in ChakraCore engine, and 2.7, 1.1, 1.9, 1.3, and 2.2 defects in the JerryScript engine, respectively. However, none of them detected defects in the

Table 2: Code Coverage.

Subject	Metric	FuzzFlow	Superion	DIE	Fuzzilli	FuzzFlowNH
SM	Average	**20.53%**	13.68%	14.27%	16.03%	19.30%
	Improvement	-	6.85%	6.26%	4.50%	1.23%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
V8	Average	**19.32%**	15.62%	13.88%	15.13%	18.41%
	Improvement	-	3.70%	5.44%	4.19%	0.91%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	0.02
JSC	Average	**22.56%**	18.38%	18.81%	19.03%	21.51%
	Improvement	-	4.18%	3.75%	3.53%	1.05%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
CH	Average	**19.52%**	16.77%	17.19%	18.44%	18.68%
	Improvement	-	2.75%	2.33%	1.08%	0.84%
	$\hat{A}_{12}$	-	0.95	0.92	0.92	0.90
	p_U	-	<0.01	<0.01	<0.01	0.01
JERRY	Average	**67.84%**	62.30%	63.55%	63.86%	65.68%
	Improvement	-	5.54%	4.29%	3.98%	2.16%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
QJS	Average	**52.05%**	40.52%	46.34%	45.13%	50.33%
	Improvement	-	11.53%	5.71%	6.92%	1.72%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
Average(among subjects)		-	5.80%	3.92%	4.62%	1.32%

V8, JavaScriptCore, SpiderMonkey, and QuickJS engines during the test period. This indicates that JS engines are adequately audited and tested. On the more vulnerable ChakraCore and JerryScript engines, FuzzFlow demonstrated the highest bug-finding capability. For industrial engines like V8, SpiderMonkey, and JavaScriptCore, dynamic testing is already integrated into their development processes. Due to extensive testing, comparing bug finding capabilities over weeks yields few bugs. Thus, our focus shifts to evaluating fuzzing based on code coverage, exemplified by Fuzzilli [20].

Branch coverage reflects the percentage of the branches exercised by the test cases over the total number of branches. A higher branch coverage implies a more thorough examination of the target's state space. For each tool and each test target, we evaluate the reached branch coverage after 24 hours of fuzzing, and count the number of test cases executed to reflect the throughput.

For FuzzFlow and Fuzzilli, we can directly obtain the number of branches triggered by the engine through the API. For AFL-based Superion and DIE, directly obtaining branch coverage is not feasible. Superion records code coverage using `1 << 20` bytes of shared memory, recording coverage in one byte per instrumented location. DIE uses one bit in shared memory to record the coverage of each instrumentation location. The shared memory used by DIE is `1 << 16` bytes. Following the method of prior work [48], we obtain the coverage recorded in the shared memory as the branch coverage explored by Superion and DIE. Montage, being a black-box fuzzer without coverage feedback, was omitted from this evaluation.

To evaluate the impact of high-level semantic nodes in FlowIR on exploring the state space of JS engines, we conducted an ablation experiment on FuzzFlow. We selected high-level semantic nodes including `TryNode`, `CatchNode` and `ForInNode`, disabled support for these nodes in FuzzFlow, and labeled this version as FuzzFlowNH. Without these nodes, FuzzFlowNH cannot process the corresponding AST nodes in the *JS2Graph* stage or generate test cases involving

these features. We then measured the code coverage achieved by FuzzFlowNH.

Table 2 presents the results of branch coverage. The row "Improvements" indicates the percentage of coverage improvement of FuzzFlow compared with baselines. Note that the comparison between Fuzzilli and Superion differs from that presented by Samuel et al. [20]. This discrepancy arises because Fuzzilli doesn't employ the initial seed set in their experimental setup. Our evaluation indicates that the mutation target and operators utilized by Fuzzilli outperform those of Superion. On all subjects, FuzzFlow achieves higher coverage than Superion, DIE, and Fuzzilli. FuzzFlow boosted average coverage by 4.78%. JS engines have extensive codebases, often exceeding tens of millions of lines. A 4.78% coverage enhancement corresponds to an increase of several hundred thousand lines. These results suggest that the bug-finding abilities of different fuzzers align closely with their coverage exploration capabilities. On average, the branch coverage of the target engines triggered by FuzzFlowNH is 1.32% lower than that triggered by FuzzFlow. This indicates that FlowIR's support for high-level JS features enhances path exploration capabilities. As FuzzFlow expands its support for these features, its ability to explore the target software's state space will also improve.

4.3 RQ3: Effectiveness of Data-Flow Mutation

The low coupling between control flow and data flow in FlowIR introduces novel mutation opportunities. Altering the data flow of the seed does not have a direct impact on the control flow, and vice versa. One immediate benefit of mutating the data-flow subgraph is the preservation of the seed's JIT compilation trigger condition, allowing mutants to continue exerting stress on the engine's JIT compiler. To demonstrate that effect, we evaluate the data-flow subgraph mutation's efficacy in upholding the seed's JIT trigger condition. We evaluate the fuzzers with a seed corpus that only contains the JS programs that trigger JIT compilation. We gather the mutants and assess their ability to consistently trigger JIT compilation. Meanwhile, we evaluate the code coverage of the JIT component with grcov[1].

Figure 5 illustrates the proportion of JIT activation by newly generated inputs. In comparison to Superion, DIE, Montage, and Fuzzilli, FuzzFlow generates 2.28×, 1.23×, 2.06×, and 1.20× the number of test cases capable of triggering JIT compilation. Moreover, FuzzFlow achieved 3.32%, 5.84%, and 9.15% higher line coverage in the SpiderMonkey engine's `js/src/jit` directory (over 330k LoC) compared to the greybox baselines Fuzzilli, DIE, and Superion, respectively. Superion and Montage may select a subtree containing a control structure when it mutates the AST, which potentially disrupts the control flow characteristics of the seed. In contrast, DIE, when selecting AST nodes for mutation, strategically filters certain control structures to focus on structural aspects. In the case of FuzzFlow's data flow mutation, the only scenario that might compromise existing JIT triggering conditions is altering the number of loops, potentially falling short of the JIT compilation threshold. In summary, FuzzFlow's data-flow subgraph mutation offers finer mutation granularity, effectively preserving triggering conditions for engine-specific functions embedded in high-quality seeds.

[1]https://github.com/mozilla/grcov

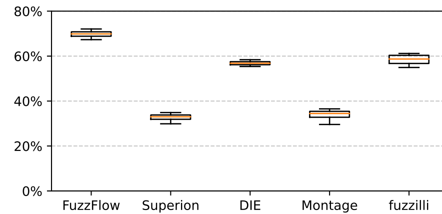


Figure 5: Proportion of tests that trigger JIT compilation

4.4 RQ4: Validity of Generated Input

Generating syntactically correct and semantically valid test cases is a prerequisite for deep code testing of the targets. A valid test case is free of syntax and semantic errors and does not trigger uncaught exceptions during execution. The higher the validity ratio of test cases generated by the fuzzer, the smaller the proportion discarded in the early stages of the engine. Upon each test case execution, the JS engine's exit code, as well as the output in stdout and stderr, indicates whether the test case has syntax or semantic errors. We run both FuzzFlow and baselines for 24 hours and record the ratio of valid generated tests over all test cases. The experiment is repeated for 10 rounds, and we used the results of statistical analysis to reduce the impact of randomness. The validity of test cases generated by Montage decreases with higher Top-k values. In our study, we set the Top-k parameter to 64, as Montage performs optimally in defect detection under this setting. Additionally, this configuration is the default in their code.

Table 3 reports the result. Overall, FuzzFlow achieves the highest test case effectiveness on all six engines. To enhance the validity, baseline fuzzers have dedicated considerable efforts beyond the mutation operators. For instance, DIE incorporates type attributes into the AST and mutates the typed AST, while Fuzzilli introduces a type system to mitigate semantic errors. Montage resolves possible reference errors by renaming them with the declared identifiers. In contrast, FuzzFlow does not impose overhead to enhance the generated test cases. It directly mutates FlowIR, achieving the desired effect. Two primary reasons contribute to the result: Firstly, FuzzFlow does not alter the syntax structure of the seed. The *Graph2JS* module generates syntactically correct test cases, thereby resolving the challenge of syntactic correctness. Secondly, FuzzFlow performs mutation within the constraints of the node type, leveraging the node's subtype functions as a JS type system. The occurrence of semantic errors is notably reduced.

4.5 RQ5: Throughput of FuzzFlow

Throughput refers to the quantity of test cases that the fuzzer mutates and executes within a given time unit. The program representation to mutate stands out as a crucial determinant influencing throughput. Given that FlowIR represents a newly developed mutation target, a key area of our focus is to assess whether it introduces substantial transformation overhead in comparison to established methodologies. We conducted a throughput comparison between FuzzFlow and baselines. The results are presented in Table 4, where throughput is measured by the number of test cases executed within 24 hours. In summary, FuzzFlow exhibits a 4.19×,

Table 3: The semantic correctness of tests generated by FuzzFlow and baselines

Subject	Metric	FuzzFlow	Superion	DIE	Montage	Fuzzilli
SM	Average	**69.18%**	39.43%	60.07%	34.38%	58.20%
	Improvement	-	29.75%	9.11%	34.80%	10.98%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
V8	Average	**72.04%**	38.96%	58.26%	35.54%	51.53%
	Improvement	-	33.08%	13.78%	36.50%	20.51%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
JSC	Average	**70.23%**	32.97%	61.83%	35.15%	50.93%
	Improvement	-	37.26%	8.40%	35.08%	19.30%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
CH	Average	**72.18%**	35.52%	56.40%	35.09%	-
	Improvement	-	36.66%	15.78%	37.09%	-
	$\hat{A}_{12}$	-	0.99	0.99	0.99	-
	p_U	-	<0.01	<0.01	<0.01	-
JERRY	Average	**64.55%**	36.52%	57.79%	38.90%	57.03%
	Improvement	-	28.02%	6.76%	25.65%	7.52%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
QJS	Average	**69.28%**	37.41%	61.95%	38.56%	63.09%
	Improvement	-	31.87%	7.33%	30.72%	6.19%
	$\hat{A}_{12}$	-	0.99	0.99	0.99	0.99
	p_U	-	<0.01	<0.01	<0.01	<0.01
Average Improvement		-	32.77%	10.19%	33.31%	12.90%

11.55× and 2.62× improvement of throughput compared to AST-based DIE, Montage, and bytecode-based Fuzzilli but falls short of the throughput achieved by Superion. The elevated throughput of FuzzFlow indicates that the mutation on FlowIR does not introduce significant additional performance overhead. This is a crucial factor for efficiently identifying defects within a specified time interval.

FuzzFlow's superior throughput is attributed to two key factors. Firstly, stored seeds in the queue remain in FlowIR format. Consequently, during the fuzzing process, each mutation only necessitates a one-way conversion from FlowIR to JS. Secondly, FuzzFlow is implemented in C++. This choice of a system-level language provides a notable efficiency advantage compared to DIE, which employs TypeScript for mutation. It is noteworthy that, in addition to observing the throughput of a fuzzer, the quality of generated test cases holds more significance. Test cases of inferior quality may fail to adequately cover the critical path of the engines. Despite FuzzFlow exhibiting a lower throughput than Superion, it is important to highlight that the validity ratio of test cases produced by FuzzFlow and the exploration of code coverage surpass those achieved by Superion to a significant extent.

4.6 Case Studies

Mutation on data-flow subgraph. Issue-261949 represents a defect identified by FuzzFlow in Safari's JavaScriptCore. Listing 3 illustrates a simplified test case that triggers the bug. The initial seed for this test case originates from Safari's regression test case `regress-176485.js`. The seed includes both `try-catch` and invocation of the `Object.defineProperty` method, encompassing two critical control-flow semantics.

FuzzFlow preserves the existing control flow within the seed, with data-flow subgraph mutation playing a pivotal role in defect

Table 4: The total number of tests executed during 24-hour fuzzing by FuzzFlow and baselines

Subject	Metric	FuzzFlow	Superion	DIE	Montage	Fuzzilli
SM	Average	656.40k	1,121k	143.58k	213.08k	337.17k
	p_U	-	<0.01	<0.01	<0.01	<0.01
V8	Average	530.25k	1,970k	134.88k	229.00k	250.32k
	p_U	-	<0.01	<0.01	<0.01	<0.01
JSC	Average	422.91k	3,218.40k	294.02k	294.3k	85.16k
	p_U	-	<0.01	<0.01	<0.01	<0.01
CH	Average	265.98k	1,688.76k	177.90k	204.25k	-
	p_U	-	<0.01	<0.01	<0.01	-
JERRY	Average	3,641.91k	18,973.78k	1,257.79k	137.85k	1,575.27k
	p_U	-	<0.01	<0.01	<0.01	<0.01
QJS	Average	7,629.24k	19,056.02k	703.82k	219.40k	4,345.47k
	p_U	-	<0.01	<0.01	<0.01	<0.01

detection. Records indicate that the applied mutation operators encompass five data-flow subgraph mutations. Notably, the first two parameters of the `Object.defineProperty` method call at line 4, the method name `assign`, and the parameters at line 6, are all derived from data flow mutation. By altering data-flow semantics through the amalgamation of independent data-flow subgraphs across the seed set, FuzzFlow successfully triggers this edge case.

```
1 var x = Object();
2 try {
3     var a = {};
4     var b = Object.defineProperty(Object, 1, a);
5 } catch(e) {}
6 var y = Object["assign"](x, Object);
```

Listing 3: Test case produced by FuzzFlow which triggers issue-261949 in JavaScriptCore

Semantic meaningful mutation. CVE-2023-5728 denotes a vulnerability identified by FuzzFlow within Firefox's SpiderMonkey. This bug specifically resides in the engine's garbage collector, stemming from an issue in the engine update process related to `weakRefMap` when a `WeakRef` target is cleared. Consequently, the test case triggers a crash upon a validity check detecting the presence of a dead wrapper within the `weakRefMap`.

FuzzFlow generates the test case in Listing 4 through effective semantic mutations. The attribute names (representing data flows in FlowIR), namely `nukeAllCCWs`, `WeakRef`, and `transplantableObject`, are sourced from three distinct seeds. It is noteworthy that the initial seed set lacks a seed containing all the aforementioned data flows. The control flows `new newGlobal` and `bar.deref` are both extracted from SpiderMonkey's regression test case `tests_gc_weakRefs.js`. The bug has existed for over three years. FuzzFlow's semantic mutation successfully merges the specified control flows and data flows in the final test case.

```
1 const g = newGlobal({newCompartment: true});
2 const domObj = this.transplantableObject().object;
3 const bar = new g.WeakRef(domObj);
4 bar.deref();
5 this.nukeAllCCWs();
```

Listing 4: Test case produced by FuzzFlow which triggers CVE-2023-5728

5 Discussion

In this section, we discuss the limitations of our approach and the possible future works.

Intra-procedural Analysis. FlowIR emerges from intra-procedural analysis, where each procedure is represented by a distinct graph. The inherent limitation of intra-procedural analysis lies in its disregard for the global context of the program, concentrating solely on dependencies within individual methods. This narrow focus may lead to the oversight of global dependencies, including those between different procedures. Inter-procedural analysis [36], on the other hand, proves adept at addressing this issue by considering dependencies between procedures and offering a holistic view of the entire program. Nonetheless, the adoption of inter-procedural analysis often comes with an associated increase in computational resources. FuzzFlow is a first but substantial step in this direction.

Bug Oracle. In fuzzing, bug oracle determines whether a given execution of the target program violates a specific security policy [33]. The most frequently employed bug oracle involves monitoring whether the input causes an execution crash. Additionally, the JS engines detect defects through internal self-checks (i.e., assertions). After the engine detects an assertion error, it will deliberately initiate a crash. However, there are still defects that may not lead to a crash. For instance, incorrect optimization by the JIT compiler might not result in memory corruption or assertion error. Improving the bug oracle can consequently enhance the effectiveness of vulnerability detection.

A commonly utilized solution is differential testing. Differential testing relies on another implementation of the exact same functionality as a reference. It involves comparing the execution outcomes of different JS engines or optimization levels, enabling the detection of non-crash defects. Differential testing has demonstrated significant efficacy in identifying JIT-related optimization defects [3] and conformance bugs [52].

The mutation target and bug oracle are two orthogonal challenges in fuzzing. The FlowIR-based mutation target outlined in this paper is adaptable and can be extended to support other testing oracles. Our upcoming research direction entails the integration of differential testing into FuzzFlow. However, these aspects are not central to the present paper. Our key contribution is introducing a graph-based IR that allows the representation of JS programs with efficient fuzzing mutators.

6 Related Work

Fuzzing JS Engines. Existing JS engine fuzzers fall into two categories based on the construction of new test cases: generation-based and mutation-based. Notable examples of generation-based fuzzers include Jsfunfuzz [35] and CodeAlchemist [21]. Jsfunfuzz is a black-box fuzzer developed by Mozilla, it creates new test cases using pre-defined grammars. CodeAlchemist learns the language semantics from a corpus of JS seed files, it extracts code bricks, and subsequently reassembling them.

The Superion study [47] shows that the mutation operator based on AST, owing to its syntax awareness, is more effective in exploring the JS engine compared to vanilla AFL. DIE [37] employs AST as its mutation target and advocates for an enhancement in the utilization of high-quality seeds through aspect preservation. Specifically, aiming to improve the testing of the JIT compiler, DIE endeavors to preserve the structure and type semantic aspects of the seed AST throughout the mutation. These two aspect features, structure and type, serve as approximations of specific control-flow structures and data flow features. Preserving these semantic features proves to be beneficial for testing deep locations within the engine. In contrast to DIE, this paper proposes the FlowIR, which directly represents control flow and data flow as a mutation target, facilitating preserving the semantic features more seamlessly.

Instead of mutating the AST, specific IRs have been proposed. Samuel et al. [20] propose to operate at bytecode level, which is closer to the engine's internal representation of the code. They introduce Fuzzilli and employ a new IR called FuzzIL to stress the JIT compiler. PolyGlot [5] is a fuzzing framework that generates high-quality test cases for processors of different programming language. To achieve the generic applicability, PolyGlot neutralizes the difference in syntax and semantics of programming languages with a uniform IR. The IR used in PolyGlot consists of a list of statements. Each statement includes an order, a type, an operator, no more than two operands, a value and a list of semantic properties. In contrast to the bytecode IR used by Fuzzilli, our FlowIR supports the direct mutation on the semantics of the seed, as the mutation target itself embodies the semantics. Compared with PolyGlot, this paper recognizes the difference (e.g., type system, memory management, concurrency) among programming languages, and therefore focuses on the IR of JS. The idea of FlowIR-based mutation is extensible to other programming languages.

Existing fuzzers have targeted specific components like binding layers or JIT compilers. Favocado [10] focuses on fuzzing binding layers of JS runtime systems. It aims to generate syntactically and semantically correct test cases and reduce the size of the input space for fuzzing. FuzzJIT [48] leverages differential testing as an bug oracle to detect non-crashing JIT compiler bugs. To facilitate each test case in triggering the JIT compilation, FuzzJIT introduces an input-wrapping template based on human knowledge. In contrast to FuzzJIT, FuzzFlow is oriented towards leveraging the inherent characteristics of the seeds themselves. Section 4.1 verifies the vulnerability detection efficacy of FuzzFlow on non-JIT components.

Mutation Operators. Mutation-based grey-box fuzzing offers distinctive advantages in detecting vulnerabilities. Combined with evolutionary algorithms, mutation-based fuzzers explore the state spaces of the target gradually. Established fuzzers like AFL and Honggfuzz [17] have pre-defined a set of mutation operators. For instance, AFL regards test cases as byte sequences and applies mutators such as byte insertion, modification, or deletion. The mutation operator stands as the core component of the fuzzer [50].

Researchers have conducted studies on the scheduling of mutation operators. MOPT [32] introduces a scheduling scheme based on the particle swarm optimization algorithm. Building upon AFL, MOPT introduces a comprehensive mutation operator scheduling algorithm designed to orchestrate operators predefined by AFL. The findings indicate that, in comparison with AFL, which selects mutation operators based on fixed probability, MOPT exhibits advantages in exploring code coverage and uncovering software defects.

However, there are limited studies on mutation operator design. The key insight of this paper is that the representation of mutation

targets influences the design of the operators significantly. Therefore, we focus on the mutation target itself and subsequently design several effective mutation operators based on it.

Graph-based Program Representation. Representing the semantics of source programs with graphs is a long-standing research problem. FlowIR differs from existing graph IR paradigms such as PDG [14, 49] and CPG [51].

PDG and FlowIR represent different control relationships. PDG consists of a Control Dependency subGraph (CDG) and a Data Dependency subGraph (DDG), with two types of edges: control dependency edge and data dependency edge. The CDG in PDG, which is designed to detect potential parallel optimization, is obtained through post-dominator analysis of the Control Flow Graph (CFG). The control dependency edges indicate that the execution of a particular statement is conditionally dependent on another. In contrast, the control flow edges in FlowIR indicate the direct flow of control from one statement to another. These two types of edges have distinct purposes in program analysis: CFG edges represent the possible paths of execution within a program, CDG edges represent the dependencies based on control conditions, specifically how the execution of certain parts of the code depends on the outcomes of conditional statements.

Converting between PDG and source code is more challenging than with FlowIR. Firstly, constructing a JS PDG is more complex than FlowIR, as it involves additional post-dominator analysis beyond the CFG. Secondly, converting a PDG back to JS is more challenging than converting from FlowIR. While no current work addresses converting PDG back to JS, we believe it is feasible. However, the reconstruction of control flow from the exact CDG is more difficult as noted in the PDG paper [14].

The CPG stands out as a popular choice for detecting vulnerabilities. A CPG integrates graph structures including AST, CFG and PDG for static analysis. The integration holds all information relevant to security analysis but is less suitable for program transformation. Firstly, subgraphs like AST and PDG contain overlapping semantic content. While this redundancy is manageable in static graphs, it complicates the design of mutation operators due to synchronization issues in dynamic contexts. For instance, mutations applied to CPG's AST necessitate the re-establishment of the CFG and PDG to preserve semantic consistency. Similarly, mutations in the PDG necessitate updates to the AST and CFG graphs, raising similar synchronization concerns. Secondly, converting the mutated composite graph back to JS source code poses another challenge. Currently, there is no existing research to guide this process. Additionally, the conversion between the mutation target and JS must be fast for effective fuzzing, but current approaches lack performance optimization. For the reasons discussed, we chose not to develop FuzzFlow based on Joern [26], despite Joern's ability to construct PDG and CPG for JS using the GraalVM JS project. Joern stores graphs in a database for static analysis applications. To the best of our knowledge, there is no research example of converting these graphs back into source code.

There are other open-source implementations of graph IR. The GraalVM IR [12] is a graph IR initially conceived for Java but has been expanded to include support for multiple languages. Both GraalVM IR and TurboFan IR [16] are derived from bytecode and subsequently optimized into machine instructions. They are more closely aligned with machine instructions, foregoing some source code-level semantics, rendering conversion back to JS unfeasible.

As described in Section 3.1, FlowIR differs from existing graph IRs in two key aspects. First, FlowIR is unique as the first graph-based IR supporting bidirectional conversion with source code, achieved through a careful redesign of nodes and edges. High-level semantics in the source code can be expressed in FlowIR, which are essential for triggering specific processing logic of target language processors. Second, we precisely define FlowIR's functional scope to avoid unnecessary information for mutations. Selecting mutation positions is a critical research topic in itself [30]. Each node and edge can be a mutation point, so redundant elements complicate effective mutation. For example, IRs in PDG paradigm, designed for parallelism detection, includes control dependency edges. These edges are crucial for its purpose but unnecessary for semantic mutation. Thus, FlowIR excludes these edges and uses control flow graphs. Unlike IRs in CPG paradigm, which includes an AST and for static vulnerability detection, FlowIR focuses on control flow and data dependency graphs to convey program semantics directly. The AST in CPG is redundant for semantic mutation, adding unnecessary complexity without benefit. We highlight the potential of graph IRs in fuzzing mutation and consider this work a first step, anticipating future advancements.

7 Conclusion

In this paper, we introduce a new graph IR to implement effective mutation operators for fuzzing JS engines. One key contribution lies in the proposal of new mutations for JS programs that are carried out on the control and data flow directly. Instead of mutating the AST or bytecode-level IR, FlowIR is developed and mutations are defined on it. Our evaluation shows that FuzzFlow achieves 18.6% higher validity of generated test cases and 4.78% higher code coverage. More importantly, FuzzFlow has found 37 new bugs in mainstream JS engines.

8 Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This project has received funding from the National Natural Science Foundation of China (grant 62402509), the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), and SNSF PCEGP2_186974. Any findings are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*.

[2] astexplorer. 2017. A web tool to explore the ASTs generated by various parsers. https://astexplorer.net

[3] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-picking: Differential fuzzing of JavaScript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 351–364.

[4] Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 400–430.

[5] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One engine to fuzz'em all: Generic language processor testing with semantic validation. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 642–658.
[6] Cliff Click and Michael Paleczny. 1995. A simple graph-based intermediate representation. *ACM Sigplan Notices* 30, 3 (1995), 35–49.
[7] Keith D Cooper and Linda Torczon. 2011. *Engineering a compiler.* Elsevier.
[8] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 25–35.
[9] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.
[10] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases.. In *NDSS*.
[11] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 110–121.
[12] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop.* 1–9.
[13] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages.* 1–10.
[14] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.
[15] Github. 2022. JavaScript stays as the 1st most used language. https://octoverse.github.com/2022/top-programming-languages
[16] Google. 2015. TurboFan is one of V8's optimizing compilers. https://v8.dev/docs/turbofan
[17] Google. 2016. Honggfuzz. https://github.com/google/honggfuzz
[18] Google. 2017. V8 features an interpreter called Ignition. https://v8.dev/docs/ignition
[19] Rahul Gopinath, Philipp Görz, and Alex Groce. 2022. Mutation analysis: Answering the fuzzing challenge. *arXiv preprint arXiv:2201.11303* (2022).
[20] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *Network and Distributed Systems Security (NDSS) Symposium.*
[21] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.. In *NDSS*.
[22] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.
[23] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, et al. 2021. SoFi: Reflection-Augmented Fuzzing for JavaScript Engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security.* 2229–2242.
[24] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12).* 445–458.
[25] Sanghoon Jeon and Jaeyoung Choi. 2012. Reuse of JIT compiled code in JavaScript engine. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing.* 1840–1842.
[26] Joern. 2021. Honggfuzz. https://github.com/joernio/joern
[27] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.
[28] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE, 2–14.
[29] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *29th USENIX Security Symposium (USENIX Security 20).* 2613–2630.
[30] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* 475–485.
[31] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. In *Proceedings of the... AAAI Conference on Artificial Intelligence.*
[32] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19).* 1949–1966.
[33] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.
[34] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering.* 1393–1403.
[35] Mozilla. 2007. A collection of fuzzers in a harness for testing the SpiderMonkey JavaScript engine. https://github.com/MozillaSecurity/funfuzz
[36] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis.* Springer.
[37] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP).* IEEE, 1629–1642.
[38] Terence Parr. 1992. ANTLR. https://www.antlr.org
[39] projectzero. 2021. CVE-2021-37975: Chrome v8 garbage collector logic bug causing live objects to be collected. https://googleprojectzero.github.io/0days-in-the-wild//0day-RCAs/2021/CVE-2021-37975.html
[40] ProjectZero. 2022. V8 0-day In-the-Wild 2021-2022. https://docs.google.com/spreadsheets/d/1lkNJ0uQwbeC1ZTRrxdtuPLCIl7mlUreoKfSIgajnSyY/view
[41] saelo. 2018. Safari RCE, sandbox escape, and LPE to kernel for macOS. https://github.com/saelo/pwn2own2018
[42] saelo. 2022. Attacking JavaScript Engines in 2022. https://saelo.github.io/presentations/offensivecon_22_attacking_javascript_engines.pdf
[43] James Stanier and Des Watson. 2013. Intermediate representations in imperative compilers: A survey. *ACM Computing Surveys (CSUR)* 45, 3 (2013), 1–27.
[44] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security.* Springer, 581–601.
[45] W3Techs. 2023. Usage statistics of JavaScript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript
[46] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP).* IEEE, 579–594.
[47] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE, 724–735.
[48] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler. (2023).
[49] Daniel Weise, Roger F Crew, Michael Ernst, and Bjarne Steensgaard. 1994. Value dependence graphs: Representation without taxation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 297–310.
[50] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering.* 1634–1645.
[51] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy.* IEEE, 590–604.
[52] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for javascript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation.* 435–450.
[53] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. {EcoFuzz}: Adaptive {Energy-Saving} Greybox Fuzzing as a Variant of the Adversarial {Multi-Armed} Bandit. In *29th USENIX Security Symposium (USENIX Security 20).* 2307–2324.
[54] Nicholas C Zakas. 2005. *Professional JavaScript for Web Developers.* John Wiley & Sons.
[55] Michal Zalewski. 2017. american fuzzy lop. http://lcamtuf.coredump.cx/afl
[56] G Zhang, P Wang, T Yue, X Kong, S Huang, X Zhou, and K Lu. 2022. Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing. In *Network and Distributed Systems Security (NDSS) Symposium*, Vol. 2022.