Detecting JavaScript Transpiler Bugs with Grammar-guided Mutation

Le Chen[a], Zhide Zhou[a], Xiaochen Li[a], He Jiang[a, b*]
[a]School of Software, Dalian University of Technology, Dalian, China
[b]the Key Laboratory for Artificial Intelligence of Dalian, Dalian 116024, China
clhiker@mail.dlut.edu.cn, cszide@gmail.com, {xiaochen.li, jianghe}@dlut.edu.cn

***Abstract*—JavaScript (JS) transpilers translate JS programs from a higher grammar standard to a lower one, which are widely used to ensure the compatibility of JS features in software (e.g., browsers). However, JS transpilers can have bugs that lead to unintended behavior in the translated JS programs. Existing JS program generation approaches could not test JS transpilers effectively since it is hard to generate a large number of valid JS programs in specific grammar standards. In this paper, we propose TransFuzz, a grammar-guided mutation approach to find JS transpiler bugs.**

The key insight of TransFuzz is to generate syntax-specific JS programs by mutating the abstract syntax trees (ASTs) of JS programs with the guidance of the specific grammar. First, TransFuzz parses JS programs collected from open-source platforms into ASTs to obtain subtrees and leaf nodes containing specific JS syntax. Then, a grammar-guided approach is developed in TransFuzz to mutate the ASTs of the given JS programs guided by different versions of JS grammar standards. In addition, mutation operations could introduce grammatical errors. To improve the correctness of the mutated ASTs, TransFuzz develops heuristic-based correction rules to correct reference errors, type errors, and syntax errors in the mutated ASTs. After correction, the mutated ASTs are converted to the corresponding JS programs. Finally, based on differential testing, TransFuzz utilizes the generated JS programs to detect JS transpiler bugs.

Our evaluation shows that TransFuzz significantly outperforms existing JS program generation approaches by triggering 47.82%–385.71% more JS transpiler bugs. Within ten months, we have reported 73 bugs on two popular JS transpilers `babel` and `swc`, of which 58 have been confirmed.

***Index Terms*—JS transpiler testing, Grammar-guided mutation, Differential testing.**

I. Introduction

JavaScript (JS) transpilers are the tools to translate JS programs written in different JS standards, which are widely used in fields such as code translation [1], [2], code obfuscation [3], [4], compiler optimization [5], [6], and static code analysis [7], [8]. JS transpilers are designed to solve JS compatibility issues. JS syntax complies with the script programming language specification ECMAScript (ES) by ECMA (European Computer Manufacturers Association) [9]. Since 2015, ECMA has updated an ES standard every year. The rapid development of ES standards brings many compatibility and security issues to the applications required to parse JS programs, since the applications may not support the latest ES standard. Fig. 1 is an example that a popular JS transpiler `babel` transpiles

```
1 input.map (item => item + 1);
2
3
```

(a) Code snippet before translation (in the ES6 standard)

```
1 input.map (function (item) {
2   return item + 1;
3 }
```

(b) Code snippet after translation (in the ES5 standard)

Fig. 1: Code snippet transpiled from ES6 to ES5 standards.

the arrow function in the ES6 standard into the anonymous function in the ES5 standard.

JS transpilers are widely used by developers. They have become an important part of browser application development toolchains. When developing browser applications, designers choose the ES development standard according to their needs. Then, they use JS transpilers to convert JS programs from a higher ES standard version into the version supported by the browser. At present, many well-known JS tools have integrated JS transpilers, such as React [10], Next.js [11], Vue [12], Ember [13], and Angular [14]. However, the correctness of JS transpilers has a great impact on the accuracy and reliability of these JS tools, since bugs in JS transpilers could inject unexpected behaviors into the transpiled JS programs. It is important to design approaches to test JS transpilers and ensure the correctness of the translation.

Currently, there have been a lot of studies about the quality assurance in terms of JS engines [15]–[18]. Some typical approaches are CodeAlchemist [19] and Montage [20]. CodeAlchemist randomly combines code snippets from different JS programs to generate new JS programs. Montage converts JS programs into ASTs; it uses deep learning to learn the AST structures, and predicts new subtree tokens to generate JS programs with different syntax structures. However, existing approaches for JS engine testing may not be applicable to JS transpiler testing since it is hard to generate effective JS programs for this task. On the one hand, when transpiling JS programs from a source ES standard (e.g., ES6) to a target ES standard (e.g., ES5), JS transpilers only process code lines containing the source ES syntax[1] (i.e., the ES6 syntax). JS programs generated by existing JS engine testing approaches may not contain plenty of source ES syntax to effectively test JS transpilers. On the other hand, JS transpilers can

*Corresponding author

[1]In this paper, we use JS syntax and ES syntax In this article, we can use JS syntax and ES syntax interchangeably.

refuse to transpiler JS programs when the programs contain grammar errors, though such programs can be accepted by JS engines if the program branches containing syntax errors are not executed. Hence, many JS programs generated by JS engine testing approaches such as deep learning cannot be used for JS transpiler testing. Hence, to effectively test JS transpilers, it is important to generate a large number of valid (i.e., with no grammar errors) JS programs with many ES syntax in the specific ES standard.

To generate such JS programs, we designed TransFuzz, a grammar-guided mutation fuzzer for JS transpiler testing. TransFuzz has four main phases, i.e., syntax-specific dataset construction, grammar-guided AST mutation, AST correction, and bug identification. TransFuzz first collects JS programs from GitHub that meet the specified ES syntax requirements. TransFuzz parses these JS programs into ASTs to collect syntax-specific subtrees and literal leaf nodes. During grammar-guided AST mutation, TransFuzz extracts an ES grammar dictionary from different versions of ES standards. Given an AST of a JS program, TransFuzz mutates the AST at the coarse-grained (subtree) and fine-grained (leaf) levels with the guidance of the ES grammar dictionary. The mutation operations follow the ES standards to ensure that the generated ASTs have the syntax features we need, thus generating a large number of syntax-specific JS programs. After AST mutation, TransFuzz corrects the generated ASTs and converts them into JS programs for JS transpiler testing. This phase corrects reference errors, type errors, and syntax errors in the mutated ASTs with a set of heuristic-based correction rules, to ensure the validity of the generated JS programs. After these steps, JS programs for JS transpiler testing are generated. For bug identification, TransFuzz uses differential testing [21] to identify syntax and semantics bugs in JS transpilers. We find syntax bugs with syntax checking tools, which determine whether transpilers completely translate the source syntax to the target syntax. Meanwhile, we compare the semantics of JS programs before and after translation. We find semantics bugs by analyzing the semantic inconsistency.

To evaluate TransFuzz, we conducted experiments for ten months on `babel` and `swc`, two of the most widely used open source JS transpilers. TransFuzz found 73 bugs, of which 58 have been confirmed. In addition, we compare TransFuzz with four state-of-the-art JS program generation approaches for JS engine testing. Experimental results show that TransFuzz significantly outperforms the baselines by up to 47.82%–385.71% in terms of the bug-finding capability on average. We find both grammar-guided AST mutation and AST correction strategies in TransFuzz positively affect the effectiveness of TransFuzz in finding bugs. The impact of the parameter of TransFuzz is also analyzed in the experiment.

The main contributions of this paper include the following:

- To the best of our knowledge, this is the first work focusing on JS transpilers testing, which can provide a new research direction for analyzing JS toolchains.
- We propose TransFuzz, an efficient testing approach, to test JS transpilers. TransFuzz found 73 bugs in real-world JS transpilers, of which 58 have been confirmed.
- We implement TransFuzz as a practical tool for JS transpiler testing.

```
var name = {
  "\uD835\uDC9C"() {}
};
```

(a) `babel` bug#13983[2]

```
const v0 = "Hello\nWorld";
console.info(`Without 's' flag:`,
  /Hello.World/.test(v0));
```

(b) `swc` bug#4192[3]

```
let v0 = (Array, ...Int32Array)
  => (NaN + Infinity) *
    Int32Array.length;
```

(c) `swc` bug#5030[4]

```
function func(...arguments) {
  console.log(arguments);
} func(1, 2, 3);
```

(d) `babel` bug#13992[5]

Fig. 2: Examples of JS transpiler bugs.

To support open science, after the paper review is over, we will upload the code and dataset to the open platform.

The rest of this paper is organized as follows. Section II introduces the background of this paper. Section III discusses the details of TransFuzz. We conduct extensive experiments to evaluate TransFuzz in Section IV. Sections V and VI introduce the threats to validity and the related work, respectively. Finally, Section VII concludes this study and future work.

II. Background

In this section, we introduce the background of JS transpilers and examples of real-world JS transpiler bugs. We also explain the challenge of finding JS transpiler bugs.

A. JS Transpilers and Their Bugs

JS transpilers are tools to translate JS programs from a higher ES standard to a lower one. We call the syntax of the program before the translation as the source syntax, and after the translation as the target syntax. In this process, JS transpilers first statically check the syntax correctness of JS programs. Programs containing syntax bugs are refused to transpile. If the syntax is correct, JS transpilers transpile code snippets in JS programs, which have the source syntax, into the target syntax required by users. The semantics of the code snippet before and after translation should remain the same. As shown in Fig. 1, the JS transpiler `babel` converts the code snippet written with the ES6 syntax into the ES5 syntax. During the translation, `babel` rewrites this code snippet to conform to the syntax features of the ES5 standard.

However, during the translation, JS transpilers may not always behave correctly. According to the definition of JS transpilers, we classify the error behavior of JS transpilers into two categories, i.e., syntax bugs and semantic bugs.

Syntax bugs are usually caused by two reasons. First, JS transpilers refuse to transpile syntactically correct code snippets (referred as “refuse translation bug”). For example,

[2]https://github.com/babel/babel/issues/13983
[3]https://github.com/swc-project/swc/issues/4192
[4]https://github.com/swc-project/swc/issues/5030
[5]https://github.com/babel/babel/issues/13992

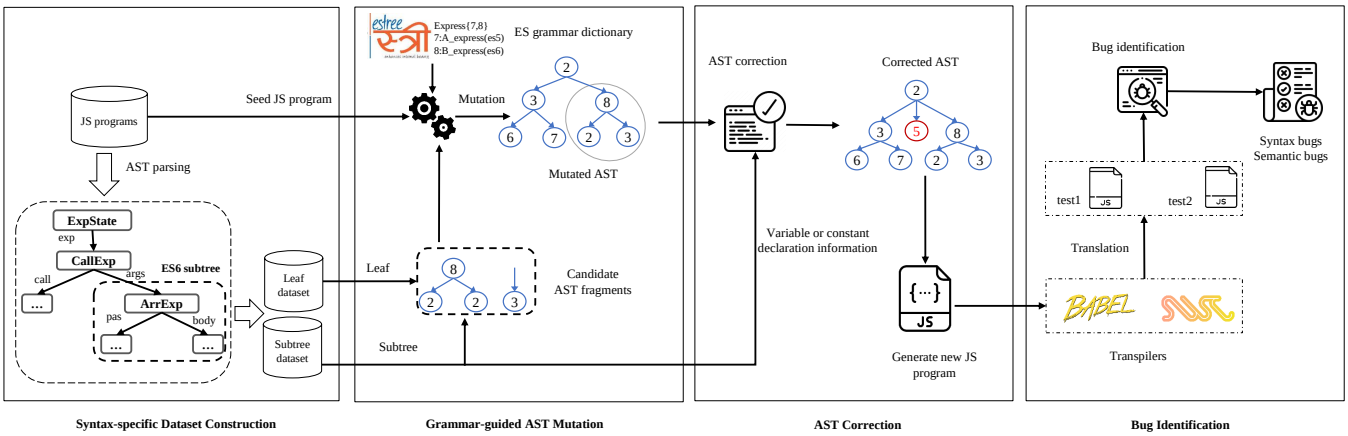

Fig. 3: Framework of TransFuzz for detecting JS transpiler bugs.

the code snippet in Fig. 2a is syntactically correct. However, it is rejected by `babel` because of the Unicode character in line 2. Another reason for syntax bugs is that the code snippet transpiled by JS transpilers does not fully conform to the target syntax features (referred as "incomplete translation bug"). As shown in Fig. 2b, the "/" label of "/Hello World/" in line 3 is the syntax "DotAll RegExp flag", which is only available after the ES9 standard; however, the JS transpiler `swc` mistakenly retains this line when transpiling the code snippet to the ES5 standard.

Semantic bugs mean the semantics of JS programs before and after translation are changed. The most common manifestation of semantic bugs is that the transpiled JS programs cannot be executed correctly (referred as "runtime error bug"). For example, when executing the transpiled code snippet in Fig. 2c with `nodejs`, the code snippet reports an error "TypeError: Array is not a constructor", though the original code snippet is correct. Another manifestation of semantic bugs is that the execution results of JS programs before and after translation are inconsistent (referred as "inconsistency bug"). As shown in Fig. 2d, the output of the code snippet is $[undefined, undefined, undefined]$ after being transformed by `babel`. However, it was $[1, 2, 3]$ before translation.

JS transpiler bugs can mislead developers. For example, a bug in the tool `react native` is caused by a decorator bug in `babel` (i.e., `babel` bug#20038). The root cause of bug#9224 in the tool `Nuxt` is due to the loose configuration bug in `babel`. A more serious bug example is the OOM problem caused by the unlimited growth of the babel-register cache (i.e., `babel` bug#5667). This bug affects multiple downstream components, including `Mochajs`, `TypeORM`, `React-static`, and `Domiii`. Since JS transpilers do not execute JS programs, developers can hardly associate any bugs in JS programs with JS transpilers. Currently, many popular front-end frameworks and tools have integrated with JS transpilers, which motivates the need to propose effective approaches for JS transpilers testing.

B. Challenge

To our knowledge, there is currently no work to test JS transpilers. One of the most relevant studies is JS engine testing. However, JS transpilers are different from JS engines. JS engines are a kind of virtual machine that are designed specifically to interpret and execute JS programs. In contrast, JS transpilers focus on transforming JS programs based on ES standards. Specifically, there are two main differences between JS engine testing and JS transpiler testing.

First, when transpiling JS programs from a source ES standard to a target ES standard (e.g., from ES6 to ES5), only code lines containing the source ES syntax (i.e., the ES6 syntax) are proceed. Although existing studies for JS engine testing [17], [19], [20], [22] aim to generate JS programs, they are not designed to generate JS programs written in specific ES syntax (e.g., ES6). As a result, most code lines generated by these approaches cannot be used to test JS transpilers.

Second, JS transpilers only accept grammatically correct programs, while JS engines can ignore errors in code snippets that are not executed. Existing approaches such as deep learning can be used to train a model with JS programs containing specific ES syntax to generate new and similar JS programs. Such JS programs may not be suitable for JS transpiler testing also, due to grammar errors. Although these grammar errors can be used for JS engine testing as long as the corresponding code lines are not executed, JS transpilers can refuse to transpiler these JS programs after the static syntax check.

Therefore, due to the rapid development of ES standards, the challenge of JS transpiler testing is to effectively generate a large number of valid JS programs that satisfy specific ES syntax features.

III. Approach

In this section, we present technical details of TransFuzz for JS transpiler testing. As shown in Fig. 3, TransFuzz has four main components, namely syntax-specific dataset construction, grammar-guided AST mutation, AST correct, and bug identification. The basic idea of TransFuzz is to

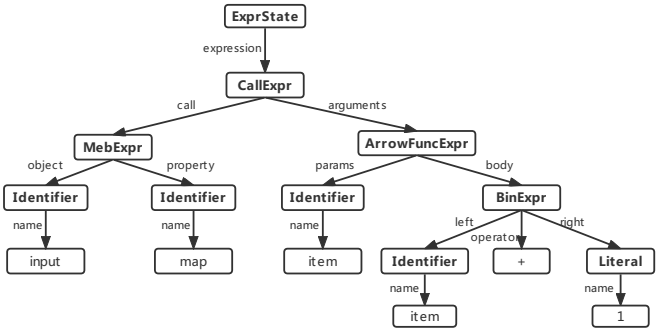

Fig. 4: An AST built for the JS code snippet in Fig. 1a.

mutate and generate JS programs with specific ES syntax based on an ES grammar dictionary and an AST-level syntax error correction.

TransFuzz first collects a set of syntax-specific JS programs and parses them as ASTs, resulting in two datasets, namely subtree dataset and leaf dataset. Next, in the grammar-guided AST mutation component, we propose a mutation technique to mutate the AST of a JS program with the guidance of ES grammar and the two datasets. This component aims to increase the ratio of specific ES syntax in the AST. All the mutated ASTs are then corrected by the AST correction component to ensure the correctness of the mutated ASTs. With these components, JS programs for JS transpiler testing are generated. In the bug identification component, we test JS transpilers using the newly generated JS programs. We compare JS programs before and after translation on both syntax and semantic levels to determine JS transpiler bugs. The output of TransFuzz is a set of JS transpiler bugs and the corresponding bug-triggering JS programs.

A. Syntax-specific Dataset Construction

Since TransFuzz detects JS transpilers by generating new JS programs using mutation techniques, we first need to construct a set of seed JS programs. We use keywords to search open-source repositories from GitHub, which contain JS programs written in different ES standards. The keywords are names of ES standards such as 'ES6' to 'ES13' and 'ES2015' to 'ES2022'. We rank the repositories by their popularity and select the top 2,000 repositories as a corpus. Next, we use `JSHint`, a static code analysis tool, to find JS programs that contain syntax in certain ES standards (e.g., ES6). We take these JS programs as seed JS programs for JS transpiler testing.

For the collected JS programs, we convert them into ASTs. For each AST, we identify and record the subtrees which are rooted with the source ES syntax features (e.g., the ES6 syntax) to create a subtree dataset. Specifically, we build a list including all declarations in the AST, such as variables, functions, and objects. We conduct preorder traversal on the AST. If a non-terminal symbol in the AST is the source ES syntax, we record the subtree rooted at this non-terminal. Next, we traverse this subtree and count all the referenced identifiers

```
1  extend interface Node <: Parents {
2      sourceType: "script" | "module";
3      body: [ Statement | ModuleDeclaration ];
4  }
```

Fig. 5: The general format of an ESTree non-terminal.

in the subtree. Finally, we construct a subtree dataset, which contains the non-terminal symbol of a subtree, the subtree itself, and the definition of referenced identifiers in the subtree.

In addition, we record the literal leaves in the AST. These literals contain user-defined content that appears in the JS programs, usually including strings, number, and regular expressions. We collect literals from the assembly, as well as manually added some special boundary literals like $None$, NaN, and $Bigint$. By the above operation, we get a leaf dataset containing literal data.

An example of our AST parsing process is presented in Fig. 4. We use `Acorn` [23], a tiny JS parser, to parse the code snippet in Fig. 1a. Fig. 4 abstractly shows the AST parsing result by `Acron`. In this AST, we keep the subtree rooted at the non-terminal `ArrowFuncExpr`, since it's a new syntax added in ES6. At the same time, we also save the declaration information of $item$, because the $item$ is referenced in this subtree. Regarding leaf nodes, we save "+" and "1" in the leaf dataset, because these literals contain user-defined content, which can be used as material for leaf node mutation.

B. Grammar-guided AST Mutation

In this subsection, we introduce the grammar-guided AST mutation phase, including ES grammar dictionary establishment and AST mutation.

1) ES Grammar Dictionary Establishment: We mutate ASTs to generate JS programs containing specific ES syntax. We guide the AST mutation by establishing an ES grammar dictionary, which contains all the available ES syntax for all ES standards. The ES grammar dictionary is used to retrieve the relationship of different ES grammar. We parse different versions of ES standards to construct the ES grammar dictionary with ESTree. ESTree [24] is a format as a lingua franca for tools that manipulate JS programs. ESTree uses custom EBNF (Backus-Naur Form) [25] syntax to describe ES syntax in ES standards. ESTree supports ES standards from ES5 to ES2022. By parsing the ESTree, we can get the relationship of different ES syntax, which is considered as the ES grammar dictionary.

Fig. 5 shows the general format of the ESTree, which represents a non-terminal syntax. In the first line, the `interface` is a keyword, and `Node` is the name of the non-terminal. Keywords `extend` and "<:" are optional. `extend` means that the current non-terminal is an extension of a previous version non-terminal, while "<:" means that the current non-terminal inherits the contents of the parent non-terminal. In between the curly braces are the rules for that non-terminal

```
interface ForInStatement
    <: Statement {
  …
}
```

(a) ES5 syntax: *ForInStatement*.

```
interface ForOfStatement
    <: ForInStatement {
  …
}
```

(b) ES6 syntax: *ForOfStatement*.

Fig. 6: An example of inheritance relationship constraint.

```
interface VariableDeclaration
    <: Declaration {
  …
  kind: "var";
}
```

(a) ES5 syntax: *VariableDeclaration*.

```
extend interface
    VariableDeclaration {
  kind: "var" | "let" | "const";
}
```

(b) ES6 syntax: *VariableDeclaration*.

Fig. 7: An example of extension relationship constraint.

(i.e., `Node`). A non-terminal can have multiple rules. Each line in curly braces in Fig. 5 represents a rule. For example, in the second line, the left side of ":" is a unique key, which is used to connect nodes in the AST. We can find these keys on the edge in Fig. 4. The right side of ":" contains multiple alternative values separated by "|". These values can be another non-terminal (e.g., *Statement* in line 3) or keywords in ES syntax (e.g., "*script*" in line 2).

2) Grammar-guided AST Mutation: To generate more JS programs for JS transpiler testing, we mutate existing JS programs collected in Section III-A. Traditional mutation tools (e.g., AFL [26]) use mutation strategies such as bit flipping, splice, and havoc, which are mostly syntactically blind. These mutation strategies are not suitable for JS transpiler testing, since they have a lower chance to generate JS programs with specific syntax (e.g., new ES syntax features in the source ES standard). To address this issue, we use a grammar-guided mutation strategy. Our mutation strategy is based on two constraints in ES standards, namely the inheritance relationship constraint and the extension relationship constraint.

(1) *Inheritance relationship constraint.* Non-terminals in ES standards have a parent-child relationship. When we select a non-terminal to mutate, the mutated non-terminal needs to inherit the same parent or grandparent non-terminal as the original non-terminal. We take the mutation of *ForInStatement* as an example to explain this constraint. In Fig. 6a *ForInStatement* in the ES5 standard inherits from *Statement*. In Fig. 6b *ForOfStatement* is a new syntax in the ES6 standard; it inherits from *ForInStatement*, which is the grandchild of *Statement*. When we mutate *ForInStatement*, we can first find the parent non-terminal *Statement*. Then, we select the descendant non-terminals of *Statement*. We find *ForOfStatement* is new syntax in the ES6 standard and satisfies the inheritance relationship constraint. Hence, we can replace the current non-terminal *ForInStatement* with *ForOfStatement* to generate a JS program with more source ES syntax.

(2) *Extension relationship constraint.* ES standards are backward compatible and extensible. New ES syntax can be easily added to the definition of existing ES syntax, either by overwriting existing rules or adding new rules. As shown in Fig. 7a and Fig. 7b, *VariableDeclaration* contains rules for different *kind* rules in ES5 and ES6 standards. *VariableDeclaration* in the ES5 standard only contains "var", while the ES6 standard extends the *VariableDeclaration* by adding "let" and "$const$". When we select a leaf node in an AST such as "var" to mutate, we can randomly choose one of "let" and "$const$" to replace the leaf to generate new JS programs with ES6 syntax.

We retrieve the aforementioned syntax relationships from the ES grammar dictionary, which are then used to guide the AST mutation of a JS program. Specifically, TransFuzz has two types of mutations, i.e., non-terminal subtree mutation and leaf node mutation.

(1) *Non-terminal subtree mutation.* We use the ES grammar dictionary to guide the mutation, which aims to generate ASTs containing specified ES syntax. Specifically, we conduct preorder traverse on the AST. Each non-terminal symbol in the AST is taken as a node. We traverse the AST, randomly select a subtree S_t rooted with the target ES syntax, and mutate it. Our mutation has two main operations:

- *Add or delete subtrees.* According to the ES standard, some non-terminals in an AST can be optional, such as nodes *Statement* and *ModuleDeclaration* in line 3 of Fig. 5. Therefore, during mutation, we can either insert or delete a subtree rooted with this kind of nodes (i.e., *Statement* or *ModuleDeclaration* nodes).
- *Replace subtrees.* We search the ES grammar dictionary for the possible source ES syntax (i.e., ES6) non-terminals that can replace the target ES5 non-terminal of S_t according to the inheritance relationship constraint. For the possible source ES syntax, we search subtrees, which are rooted with such syntax, in the subtree dataset. We randomly select one of the subtrees to replace S_t.

Algorithm 1 Non-terminal subtree mutation (replace subtree)

Input: an AST for mutation, subtree dataset $database$, ES grammar dictionary $model$, maximum trials $maxTrial$
Output: Mutated AST

```
1:  iter = 0
2:  while iter < maxTrial do
3:      esTarNode = randFindTarPos(AST)
4:      parNonTerm = getParent(model, esTarNode)
5:      if parNonTerm == None then
6:          iter ++
7:          continue
8:      else
9:          dscNonTerms = getDSC(model, parNonTerm)
10:         newNonTerm = random(dscNonTerms)
11:         newSub = getSubtree(newNonTerm, database)
12:         replaceSub(AST, esTarNode, newSub)
13:         break
14:     end if
15: end while
```

Algorithm 1 specifies the subtree replacement process. We first initialize the number of iterations *iter* and the maximum number of trials *maxTrial* for finding non-terminal positions of source ES syntax (e.g., ES5) for mutation (lines 1–2). We use function *randFindTarPos* to randomly select a non-terminal related to the target ES syntax from the AST (line 3). Next, we look up the parent node of this non-terminal by *getParent* function. If the node has no parent node, we go back to line 3 to find a new position in lines 3–7. If we fail to find a position after *maxTrial* trials, we stop mutating this AST. When the position is found, we search the ES syntax of the parent node through the ES grammar dictionary. We use function *getDSC*, constrained by the inheritance relationship, to find descendant non-terminals to satisfy the source ES syntax (e.g., ES6). We randomly select one non-terminal source ES syntax *newNonTerm* (lines 8–10). For the searched target ES syntax, we turn to the subtree *dataset* to randomly select a subtree rooted with the searched *newNonTerm*, and use this subtree to replace the subtree at position *esTarNode* (lines 12–15).

(2) *Terminal leaf mutation.* Terminal leaf mutation is also grammar-guided. As shown in Algorithm 2, we search the AST and use the function *getSon* to find a node with no child (i.e., a terminal leaf) (lines 2). We first use function *isLiteral* to determine whether the node is a literal leaf (line 3). For literal leaves, such as string, number, and boolean, we randomly select a literal from the leaf dataset built in Section III-A (line 4). For non-literal leaves, they are usually keywords, we use the function *getLeaf* to find new *keywords* terminals that are extended in the source ES standard (e.g., ES6) based on the extension relation constraint, and randomly get one of the new terminals by function *randGetLeaf* (lines 6–7). For example, when we find a node such as *VariableDeclaration*, we can find three options (i.e., "var", "let", and "$const$" as shown in Fig. 7b) by querying the ES grammar dictionary. We randomly select one of options "let" and "$const$" as a leaf, since they are only in the source ES syntax (i.e., ES6). When we have a *newNode*, we can use it to replace the selected *node* (lines 10).

Based on the mutation operations, TransFuzz generates new ASTs with more source ES syntax. TransFuzz first mutates an AST with the non-terminal subtree mutation. We randomly add, delete, or replace a subtree in the AST. After this, we continue mutating the AST with the terminal leaf mutation. We treat a round of non-terminal and terminal mutations as a complete mutation process. To increase the complexity of the mutated AST, TransFuzz repeats the complete mutation process p times (a parameter) to generate the final AST.

C. AST Correction

Mutation operations can introduce grammatical errors. The main errors in the AST mutation of TransFuzz are *reference errors*, *type errors*, and *syntax errors*. First, mutation operations assemble non-terminal and terminal nodes in different contexts, which brings reference errors. Second, the complex expression we call may contain undefined structures, causing type errors. Third, the ES grammar dictionary we build does not include the complete ES standard. The reason is, when taking into the complete ES standard to build an ES grammar dictionary, its difficulty is equivalent to rewriting a JS engine. The missing of some special syntax (e.g., 'await expression must be in async function') in the ES grammar dictionary can lead to syntax errors. Since JS transpilers can reject JS programs with grammatical errors, we use AST correction to resolve these three types of errors to increase the number of corrected JS programs for JS transpiler testing.

Algorithm 2 Terminal leaf mutation

Input: an AST for mutation, the leaf dataset $leaves$, ES grammar dictionary $model$, ES standard ESs
Output: Mutated AST

```
1: for node in AST do
2:   if node.getSon() == None then
3:     if isLiteral(model, node) then
4:       newNode = random(leaves, node)
5:     else
6:       keywords = getLeaf(model, node, ESs)
7:       newNode = randGetLeaf(keywords)
8:     end if
9:   end if
10:  changeLeaf(AST, node, newNode)
11: end for
```

Reference errors. Regarding variable references, JS has two key features, i.e., weak typing and declaration hoisting. Weakly typing means variable types in JS can be automatically inferred. Declaration hoisting is that JS variables are usually valid within a certain scope; functions, variables, and declarations can be hoisted to the very top of the scope. Based on these two features. we collect all referenced but undeclared variables, function names, and class names in one scope. We declare these objects at the top of the scope. We add declarations for subtrees that are added or replaced when mutating, because they may refer to variables that are not declared in the current program. We also keep declarations of replaced or deleted subtrees, since they may be referenced elsewhere in the program. The former is saved when we store a subtree in Section III-A. For the latter, we statically parse a replaced subtree and extract the declared subtrees within it. We add declarations to the top of the scope.

Type errors. We check function calls and complex expressions in the ASTs. We resolve these types of errors by adding an Object structure to the program header. The properties of the structure are defined according to the references in the code.

Syntax errors. For syntax errors, we process each special syntax differently by modifying specific parameters in the AST. For example, *YieldExpress* in JS can only appear in generative functions. In the AST, this grammatical feature can be expressed as that, if the subtree of a function declaration contains the non-terminal *YieldExpress*, we need to set the "generator" leaf as "TRUE" to startup the generative expression. Therefore, to resolve syntax errors caused by missing generator function, we can traverse the AST and set the

"generator" leaf nodes of all function declaration subtrees containing *YieldExpress* to be "TRUE".

After AST correction, we use `escodegen`, a third-party tool to generate code from ASTs, to convert ASTs into JS programs [27]. We check the syntax and semantics of the generated JS programs with tools `JSHint` and `nodejs`, respectively. We only keep JS programs that can pass the two checks.

D. Bug Identification

Since the output of JS transpilers is source code, we use differential testing to determine whether the translation result contains bugs.

We take JS programs corrected in Section III-C as testcases to test JS transpilers. When a bug is found, we record the error information and the corresponding JS programs. Since our JS programs have been checked using tools `JSHint` and `nodejs` before translation, the reported errors are likely to be caused by JS transpiler bugs. TransFuzz mainly collects two types of bugs, i.e., syntax bugs and semantic bugs.

For syntax bugs, we use `JSHint` to check every successfully transpiled JS program. If the transpiled JS program contains non-target syntax or wrong syntax, it means JS transpilers cannot complete the translation caused by the potential bugs of JS transpilers. We report these potential bugs to developers after analysis.

For semantic bugs, we check whether the semantics of JS programs before and after translation is changed. The most common situation is that the transpiled JS program gets an error when being executed by JS interpreters. In this case, it is obvious that the JS transpiler has introduced a bug to the JS program. Another more obscure situation is that the transpiled program can be executed correctly. However, when we compare the execution results of the program before and after translation, results are inconsistent. This case also indicates a JS transpiler bug. To identify semantic bugs, we use `nodejs` to execute JS programs before and after translation. `nodejs` is developed based on Google v8 JS engine, which supports the latest ES standard. If a transpiled JS program cannot be executed by `nodejs`, we transpile the same JS program with another JS transpiler. When the execution results are inconsistent, we manually analyze the reason of the semantic inconsistency. If the inconsistency is not caused by random behaviors (such as *random*) in the code snippet, we consider it as a bug.

The output of this step is a set of JS transpiler bugs and the corresponding source JS programs.

IV. Evaluation

To assess the effectiveness of TransFuzz, we conducted experiments based on the following research questions (RQs).

- **RQ1:** Is TransFuzz effective in finding bugs for real-world JS transpilers?
- **RQ2:** What is the effectiveness of TransFuzz compared with existing JS program generation approaches?
- **RQ3:** What is the impact of each component on the effectiveness of TransFuzz?
- **RQ4:** What is the effect of the parameter of TransFuzz?

TABLE I: Bugs found by TransFuzz on the latest development versions of `babel` and `swc`

	Syntax bugs		Semantic bugs		Invalid or duplicate	Total
	Refuse translation	Incomplete translation	Runtime error	Inconsistency		
babel	5	0	8	5	11	29
swc	15	5	10	10	4	44

RQ1 evaluates the ability of TransFuzz to find bugs in real-world JS transpilers (i.e., two popular JS transpilers `babel` and `swc`). RQ2 focuses on the effectiveness of existing JS program generation approaches in discovering JS transpiler bugs, compared with TransFuzz. RQ3 verifies the effectiveness of the main components in TransFuzz. RQ4 evaluates the influence of the parameter in TransFuzz.

TransFuzz is implemented with 3,000 lines of code in Python 3.8. As for the conversion of JS programs and ASTs, we use the third-party tools `Acorn` and `escodegen`, respectively. In addition, we use the official default configuration files for `babel` and `swc`, which are configured to convert programs written in the ES6 standard and above (named as ES6plus) into ES5 programs.

We collected 2,000 repositories, which have JS programs containing the ES6plus syntax from GitHub with the process in Section III-A. The preprocessed corpus contains a total of 22,473 JS programs, each of which contains one or more code snippets with ES6plus syntax features. This corpus is used to construct the subtree dataset and the leaf dataset.

Our experiment was performed on a Linux server with Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz (32 cores 64 threads) and 256GB RAM, running on Ubuntu 20.04 (x86_64).

A. Answer to RQ1: Effectiveness of TransFuzz

To assess the ability of TransFuzz on finding real-world JS transpiler bugs, we conducted an experiment over ten months from November 2021 to July 2022. We mainly test the latest development version of `babel` and `swc`. The time of one testing process is about three days since the development version of JS transpilers is frequently updated[6].

We submit bugs found by TransFuzz to developers for confirmation. Given a bug, we first search for bugs previously found and bugs in the bug repository of JS transpilers according to the keywords in the error message. We manually analyze the top search results to de-duplicate bugs. In addition, we reduce the bug-triggering JS programs to facilitate developers for quick bug confirmation. We decompose a JS program into a set of code segments and delete the code segment from the AST one by one. After deletion, we convert the AST back to a JS program. If the trimmed program can still trigger the same

[6]The testing process is not continuous since we have to occasionally resolve deployment and hardware issues.

```
const Events = /\P{Number}/s; // Res is not defined
console.log(Events)
```

Fig. 8: JS transpiler refuses to translate correct code[7].

```
v3 = String.raw(
    { raw: 'abcd' },
    ...'___'); // ... still keep after transpiling
 console.log(v3);
```

Fig. 9: Transpiled JS program contains source ES syntax[8].

bug, we repeat the above process until the reduced JS program cannot trigger the bug. After the above operations, we get the reduced JS program, which is submitted to developers.

As shown in Table I, TransFuzz finds a total of 73 bugs, including 29 bugs for `babel` (18 confirmed as unique bugs) and 44 bugs for `swc` (40 confirmed as unique bugs). A total of 15 bugs were flagged by developers as invalid or duplicate bugs. In the experiment, TransFuzz finds more bugs in `swc` than in `babel`. The reason is that `swc` is a project launched in 2020. Such a younger project could have more bugs. In addition, TransFuzz finds many semantic bugs in the two JS transpilers (i.e., 13 in `babel` and 20 in `swc`). Such bugs are very important to find, since semantic bugs can change the semantics of JS programs after translation. It could be difficult for developers to locate these bugs, as developers have to be familiar with the source code of JS transpilers. Hence, a tool like TransFuzz is important for JS developers to finds these bugs.

We introduce some examples of the four types of bugs found by TransFuzz.

Refuse Translation: An example of this type of bug is `babel` bug#1485. The bug has been confirmed by developers. Fig. 8 shows the POC (proof of concept) code snippet. The first line of the code snippet, when transpiled by `babel`, triggers an error: "Res is not defined". However, the code snippet itself is correct, which can be executed by `nodejs`. This bug comes from `babel`'s dependency package `regjsparser` 0.8. Although the bug has been fixed in `regjsparser` 0.9, the developers of `babel` did not update the package. Our bug-triggering JS program is generated by the fine-grained mutation of leaf nodes of the AST for a JS program. TransFuzz uses a randomly selected regex value leaf to replace the original number value, which triggers this bug.

Incomplete Translation: Fig. 9 shows a bug reported as `swc` bug#4190. In the figure, the spread operator "..." is a new syntax of ES6. Unfortunately, the code snippet translated by `swc` still retains the spread operator. The error came from the `es-compat` package. Developers fixed this bug on the same day as we reported it. This bug-triggering JS program comes from our non-terminal subtree mutation, in which the subtree rooted at *SpreadElement* is added to the arguments list of *CallExpression*.

```
(() => {  // After transpiling is function ()
    array[args]++;
    return;
})().map(c, v0).map++;
```

Fig. 10: Transpiled JS program reports an error[9].

```
class Person {
// instanceof does not cast [Symbol.hasInstance] call to boolean
    static [Symbol.hasInstance]() {}
}
let o = {}
console.log(o instanceof Person) //print undefined but shold be false
```

Fig. 11: Results before and after translation are inconsistent[10].

Runtime Error: An example of this type of bug is `babel` bug#14401. The brief POC is shown in Fig. 10. When we check the transpiled code snippet by `nodejs`, it reports an error: "Function statements require a function name". This bug comes from the `babel`'s generator. For anonymous functions in parentheses in the figure, `babel` does not conduct the translation. Since `babel` directly removes the parentheses during printing, it causes the function in parentheses to report an error. This bug was marked as a good first issue by developers. TransFuzz replaces *CallExpression* on the AST subtree with an anonymous *ArrowFunctionExpression*, and generates the JS program.

Inconsistency: Fig. 11 shows an example of this type of bug (i.e., `swc` bug#2836). The original code snippet outputs "false" after running through `nodejs`; however, the transpiled code snippet outputs "undefined". The reason of this bug is that the method *helper.instanceof* does not cast *Symbol.hasInstance* call to boolean. We find this bug by our non-terminal subtree mutation, which mutates an existing subtree with the static method subtree *Symbol.hasInstance*.

Conclusion. TransFuzz reports 73 bugs on two real-world JS transpilers, of which 58 are confirmed. The experimental results show that TransFuzz is effective at finding bugs in real-world tools.

B. Answer to RQ2: Comparison with Baselines

In this RQ, we compare the number of bugs found by TransFuzz with four state-of-the-art JS program generation approaches, which are used to test JS engines. They are CodeAlchemist, DIE, Montage, and JEST. CodeAlchemist and Montage use code snippet composition and deep learning to generate JS programs, respectively. DIE is an AFL-based mutation fuzzer that mutates JS programs at the AST level. JEST

[7]https://github.com/babel/babel/issues/14857
[8]https://github.com/swc-project/swc/issues/4190
[9]https://github.com/babel/babel/issues/14401
[10]https://github.com/swc-project/swc/issues/2836

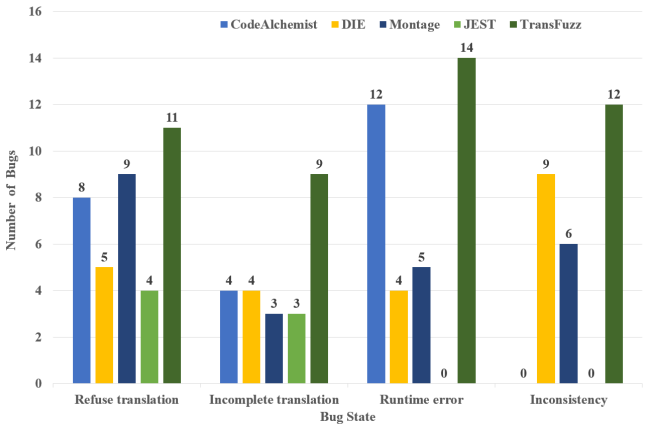

Fig. 12: Bugs found by TransFuzz and baselines in 72 hours.

is a JS program generator, which can automatically generate JS programs based on the ES grammar model extracted from ES standards. Among these four approaches, CodeAlchemist and JEST are generation-based algorithms, while DIE and Montage mainly focus on JS program mutation.

Our experiment is conducted on old released versions of `babel` 7.14.8 and `swc` 1.2.103, such that we can find a statically significant number of bugs for comparisons. We execute each approach 72 hours. When a JS program is generated, we use the bug identification strategy in Section III-D to find bugs. For a detected bug, if it does not exist in the latest development version, we take the bug as a real fixed bug; otherwise, we report it to developers for confirmation.

Fig. 12 shows the number of bugs found by TransFuzz and baselines. TransFuzz finds more bugs than all state-of-the-art JS fuzzers during the testing period. TransFuzz finds a total of 46 bugs, while CodeAlchemist, DIE, Montage, and JEST find only 24, 22, 23, and 7 bugs, respectively. Besides, the bugs found by TransFuzz cover the other four tools, and 12 bugs are the only ones found by TransFuzz. These four baselines differ in their ability to find different types of bugs.

CodeAlchemist has a good effect on finding runtime error bugs, but it has not found inconsistency bugs. In contrast, DIE finds more inconsistency bugs than runtime error bugs. After analysis, we find that CodeAlchemist usually introduces more complex syntax structures, which makes it find many runtime bugs. Regarding DIE, DIE's mutation retains the subtle syntax structure and type information in the JS program. Most of the mutated program context is preserved, but the syntax structure is not fully mutated. Hence, it finds many inconsistency bugs. Montage uses deep learning to generate JS programs. Although it can find bugs across the four categories, the number of bugs found is limited by the high number of invalid programs with grammar errors. JEST only finds 7 syntax bugs. The reason is the input of JEST is the ES standard, and the output is a piece of code snippets. JEST does not generate JS programs with complex structures to trigger semantic bugs.

For TransFuzz, the input of TransFuzz includes the original JS corpus and the ES grammar dictionary. It retains the context of the seed JS program based on introducing the source syntax structure and surpasses the above four baselines in the ability to find syntax and semantic bugs.

Conclusion. TransFuzz finds in total 46 bugs of different types in 72 hours of testing, outperforming the baselines by up to 47.82%–385.71% in terms of the bug-finding capability. TransFuzz significantly outperforms state-of-the-art approaches for JS transpiler testing.

TABLE II: Bug statistics of TransFuzz and its variants.

	Min. bugs	Max. bugs	Avg. bugs	p-value
TransFuzz$_{NC}$	15	24	20	<.001
TransFuzz$_{NG}$	19	25	22.5	.002
TransFuzz	21	30	26.1	-

C. Answer to RQ3: Impact of Different Components

Grammar-guided AST mutation and AST correction are the main components of TransFuzz for generating JS programs with specific ES standards. These components first mutate the AST of a JS program with the guidance of an ES grammar dictionary and then correct the mutated AST to generate the final JS program for testing. To access the impact of these components, we design two variants of TransFuzz, i.e., TransFuzz$_{NG}$ and TransFuzz$_{NC}$. TransFuzz$_{NG}$ conducts AST mutation without the guidance of the ES grammar dictionary established in Section III-B1. TransFuzz$_{NC}$ removes the AST correction step, which directly uses JS programs after the grammar-guided AST mutation for testing. We use TransFuzz and the two variants to test the same versions of `babel` and `swc` used in RQ2. We conduct the experiment 10 times. Each time we execute these approaches for 12 hours on the two JS transpilers, and analyze the number of bugs they found.

Table II presents the experiment result. The second to third columns are the minimum and the maximum number of bugs found by TransFuzz and its variants respectively over the ten times of experiments. The fourth column is the average number of bugs they found. TransFuzz finds a higher maximum, minimum, and average number of bugs than the other two variants. We performed a Mann-Whitney U-test for the total number of bugs found by TransFuzz and variants. The p-values show that TransFuzz is significantly better than TransFuzz$_{NC}$ and TransFuzz$_{NG}$. In addition, we find that TransFuzz$_{NG}$ seems to be better than TransFuzz$_{NC}$. The reason is that TransFuzz$_{NC}$ has no AST correction step. A large number of mutated JS programs are rejected by the JS transpilers due to syntax errors.

Conclusion. The experimental results show that TransFuzz outperforms TransFuzz$_{NG}$ and TransFuzz$_{NC}$, which indicates that the proposed grammar-guided AST mutation and AST correction techniques can improve the efficiency to detect JS transpiler bugs.

D. Answer to RQ4: The Impact of the Parameter

Our grammar-guided mutation is conducted iteratively, that is, the AST of a JS program is mutated by TransFuzz several

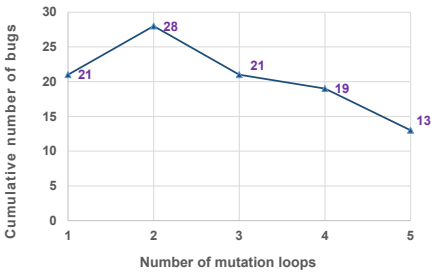

Fig. 13: Impact of different loops of complete mutation.

times iteratively until it reaches a certain threshold p (as explained in Section III-B2). Through multiple iterations, we increase the complexity of the mutated AST, thereby increasing the ability of TransFuzz to find JS transpiler bugs. To access the impact of the parameter p, we use TransFuzz to test JS transpilers by setting p as 1 to 5. Other settings are the same as those of RQ2.

Fig. 13 shows the relationship between the number of iterations p and the number of detected bugs. The results show that a small number of iterations can increase the number of bugs found by TransFuzz. However, too many iterations can reduce the bug-finding capability. TransFuzz has the best performance when the number of iterations is 2. Although iterative mutation is beneficial to improve the ability of TransFuzz to find bugs, too many iterations also increase the errors in the mutated AST. As a result, the AST correction step in Section III-C may have difficulty in getting the correct AST for testing. When the number of iterations is 2, the complexity and the success rate of mutated ASTs could have the best balance.

Conclusion. Experimental results demonstrate that the iteration number to mutate AST can affect the efficiency of TransFuzz. When the iteration number is 2, TransFuzz can obtain better results.

V. Threats to Validity

The first threat is that JS programs generated by TransFuzz may contain indeterminate behavior, though we take steps in Section III-D to mitigate these situations. However, indeterminate behaviors of JS are diverse, such as code snippets that can print stack information at runtime. In this case, since the code uses a different memory address each time it runs, the stack information will change each run. For such JS programs, we can only manually analyze whether they are valid JS programs that trigger JS transpiler bugs.

Secondly, JS transpilers have become a rich tool set after years of development. For example, `babel` includes functions such as translation, parsing, and compression. Although the transpiler we tested is a core function of the JS transpiler, it is still not enough to say that the transpiler is fully tested. In addition to the traditional ES syntax translation, the JS transpiler also provides JSX, React, Flow, and TypeScript for universal JS program translation [11], [28]–[30]. Since most of these features are still in development, we take the testing of these features as future work.

VI. Related Work

Our work is closely related to the work on JS engine testing.

The early work of JS engine testing focused on testing the interpreter of the JS engine. They mainly found bugs related to syntax analysis and shallow semantic analysis. For example, the open source project Dharma [31] used code generation and grammar-based mutation techniques to find bugs in the JS engine parser. Holler et al. [15] used the well-known language parser ANTLR4 to guide the mutation to fuzz the interpreter.

Wang et al. [32] used the knowledge in a large number of existing JS program samples to generate evenly distributed seed JS programs for fuzzing.

With the continuous popularity of the JS language, researches on JS engine testing have further developed. The testing of the JS engine has gradually developed from the parser testing to the discovery of deeper memory defects, JIT testing, etc. Li et al. [33] proposed a cost-effective dynamic update algorithm based on AFL. Their approach could generate more efficient JS seed programs. Wang et al. [34] introduced a syntax-aware adjustment strategy to improve the code coverage of AFL. Han et al. [19] noticed the dynamic characteristics of JS programs. They proposed a JS program generation algorithm of semantic aware assembly. Aschermann et al. [35] used syntax code coverage as feedback to guide JS engine testing, which performance was even one order of magnitude better than AFL. Lima et al. [18] noticed the incompleteness of ES standards caused by the rapid development of JS. They used test transplantation and differential testing to find functional errors in JS engines. Park et al. [16] found that existing techniques do not fully utilize the JS program corpus. They preserved the subtle semantics encoded in the corpus during the seed mutation process, thus touching deeper bugs in JS engines. Park et al. [36] also proposed an N + 1 version differential testing approach, which found bugs in JS engines. Dinh et al. [37] focused on testing the programming interface of JS runtime systems with JS program generation techniques.

With the development of artificial intelligence, Lee et al. [20] took the ASTs of JS programs as inputs. They generated effective JS programs by using the short-term memory model (LSTM). Ye et al. [17] used the latest pre-trained model GPT-2 to further improve the syntax accuracy of the generated JS programs. Tolksdorf et al. [22] found bugs in the JS engine debugger through the method of differential testing with machine learning.

Our work is different from JS engine testing. These approaches may not be applied for testing JS transpilers. The main reason is that JS transpilers focus on the conversion of specific ES syntax, while JS programs generated by the JS engine testing are extensive and comprehensive. Such JS programs do not focus on the generation or mutation of JS programs with specific ES syntax. According to the evaluation in RQ2, TransFuzz outperforms several state-of-the-art JS engine testing approaches.

VII. Conclusion and Future Work

In this paper, we proposed TransFuzz, an automated testing framework based on grammar-guided mutation, for JS transpiler testing. TransFuzz generates JS programs with specific syntax through grammar-guided mutation and collects different kinds of bugs according to the characteristics of JS transpilers. Evaluations show that TransFuzz is effective at finding bugs in real-world tools. We have reported 73 bugs on two real-world JS transpilers, of which 58 bugs have been confirmed by developers as unique bugs. Besides, TransFuzz significantly outperforms existing state-of-the-art approaches by detecting 47.82% to 385.71% more bugs on average. For future work, we consider extending JS transpiler testing to JS completeness testing, including testing for tools like JS transpiler, beautification, compression, and obfuscation. Another possible direction is to test transpilers for other programming languages.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 62032004, 62202079), and the Fundamental Research Funds for the Central Universities (No.DUT22RC(3)028).

References

[1] Wolfgang Christian, Mario Belloni, Robert M Hanson, Bruce Mason, and Lyle Barbato. Converting physlets and other java programs to javascript. *The Physics Teacher*, 59(4):278–281, 2021.

[2] Yogesh Maheshwari and Y Raghu Reddy. A study on migrating flash files to html5/javascript. In *Proceedings of the 10th Innovations in Software Engineering Conference*, pages 112–116, 2017.

[3] Bassam Sayed, Issa Traoré, and Amany Abdelhalim. If-transpiler: Inlining of hybrid flow-sensitive security monitor for javascript. *Computers & Security*, 75:92–117, 2018.

[4] Ukko Sarekoski et al. Transpiler-based architecture for web applications. 2021.

[5] Micha Reiser and Luc Bläser. Accelerate javascript applications by cross-compiling to webassembly. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, pages 10–17, 2017.

[6] Mateusz Bysiek, Aleksandr Drozd, and Satoshi Matsuoka. Migrating legacy fortran to python while retaining fortran-level performance through transpilation and type hints. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 9–18. IEEE, 2016.

[7] Shahriar Rostami Dovom. *Identification Of JavaScript Function Constructor Using Static Source Code Analysis*. PhD thesis, Concordia University, 2016.

[8] Dávid Honfi and Gábor Szárnyas. Static analysis algorithms for javascript.

[9] ECMA Ecma. 262: Ecmascript language specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*, 1999.

[10] Sanchit Aggarwal. Modern web-development using reactjs. *International Journal of Recent Research Aspects*, 5(1):133–137, 2018.

[11] Mohit Thakkar. Next. js. In *Building React Apps with Server-Side Rendering*, pages 93–137. Springer, 2020.

[12] Katina Kyoreva. State of the art javascript application development with vue. js. In *Proceedings of International Conference on Application of Information and Communication Technology and Statistics in Economy and Education (ICAICTSEE)*, pages 567–572. International Conference on Application of Information and Communication ..., 2017.

[13] Jesse Cravens and Thomas Q Brady. *Building Web Apps with Ember. js: Write Ambitious Javascript*. " O'Reilly Media, Inc.", 2014.

[14] Brad Green and Shyam Seshadri. *AngularJS*. " O'Reilly Media, Inc.", 2013.

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

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

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

[18] Igor Lima, Jefferson Silva, Breno Miranda, Gustavo Pinto, and Marcelo d'Amorim. Exposing bugs in javascript engines through test transplantation and differential testing. *Software Quality Journal*, 29(1):129–158, 2021.

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

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

[21] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[22] Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. Interactive metamorphic testing of debuggers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 273–283, 2019.

[23] Marijn Haverbeke. A small, fast, javascript-based javascript parser, 2020.

[24] Junliang Huang. Nicholas C. Zakas, Ingvar Stepanyan. The estree spec. https://github.com/ESTree/ESTree.

[25] Niklaus Wirth. Extended backus-naur form (ebnf). *Iso/Iec*, 14977(2996):2–21, 1996.

[26] Michal Zalewski. American fuzzy lop, 2017.

[27] Yusuke Suzuki. Escodegen, 2015.

[28] Pratik Sharad Maratkar and Pratibha Adkar. Re act js-an emerging frontend javascript libr ary. *Iconic Research And Engineering Journals*, 4(12):99–102, 2021.

[29] Akshat Paul and Abhishek Nalwaya. The ecosystem: Extending react native. In *React Native for Mobile Development*, pages 225–232. Springer, 2019.

[30] Steve Fenton, Fenton, and Spearing. *Pro TypeScript*. Springer, 2014.

[31] Christoph Diehl. Generation-based, context-free grammar fuzzer. https://github.com/MozillaSecurity/dharma.

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

[33] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 533–544, 2019.

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

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

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

[37] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases. In *NDSS*, 2021.