SoFi: Reflection-Augmented Fuzzing for JavaScript Engines

Xiaoyu He[1,2], Xiaofei Xie[3,*], Yuekang Li[3], Jianwen Sun[4], Feng Li[1,2,*], Wei Zou[1,2], Yang Liu[3], Lei Yu[1,2], Jianhua Zhou[1,2], Wenchang Shi[5], Wei Huo[1,2]

[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[4]Software Engineering Application Technology Lab, Huawei Technologies, Shenzhen, China
[5]Renmin University of China, Beijing, China

ABSTRACT

JavaScript engines have been shown prone to security vulnerabilities, which can lead to serious consequences due to their popularity. Fuzzing is an effective testing technique to discover vulnerabilities. The main challenge of fuzzing JavaScript engines is to generate syntactically and semantically valid inputs such that deep functionalities can be explored. However, due to the dynamic nature of JavaScript and the special features of different engines, it is quite challenging to generate semantically meaningful test inputs.

We observed that state-of-the-art semantic-aware JavaScript fuzzers usually require manually written rules to analyze the semantics for a JavaScript engine, which is labor-intensive, incomplete and engine-specific. Moreover, the error rate of generated test cases is still high. Another challenge is that existing fuzzers cannot generate new method calls that are not included in the initial seed corpus or pre-defined rules, which limits the bug-finding capability.

To this end, we propose a novel semantic-aware fuzzing technique named SoFi. To guarantee the validity of the generated test cases, SoFi adopts a fine-grained program analysis to identify available variables and infer types of these variables for the mutation. Moreover, an automatic repair strategy is proposed to repair syntax/semantic errors in invalid test cases. To improve the exploration capability of SoFi, we propose a reflection-based analysis to identify unseen attributes and methods of objects, which are further used in the mutation. With fine-grained analysis and reflection-based augmentation, SoFi can generate more *valid* and *diverse* test cases. Besides, SoFi is general in different JavaScript engines without any manual configuration (*e.g.*, the grammar rules). The evaluation results have shown that SoFi outperforms state-of-the-art techniques in generating semantically valid inputs, improving code coverage and detecting more bugs. SoFi discovered 51 bugs in popular JavaScript engines, 28 of which have been confirmed or fixed by the developers and 10 CVE IDs have been assigned.

CCS CONCEPTS

• **Security and privacy → Systems security**.

KEYWORDS

fuzzing, security, vulnerability

ACM Reference Format:
Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, Wei Huo. 2021. SoFi: Reflection-Augmented Fuzzing for JavaScript Engines. In *Pro-ceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460120.3484823

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484823

1 INTRODUCTION

JavaScript engines are designed to interpret JavaScript code that is usually used in web applications. In addition to being used in Web browsers, JavaScript is also included in other applications. For example, the JavaScript engine MuJS is embedded in PDF-related office software [10], XS is embedded in Kindle, Apple or Android applications [16], and mJS [3] is applied in IoT devices for running JavaScript applications.

JavaScript interpreters are particularly prone to security issues. Until 2019, more than 2300 bugs have been discovered in JavaScript engines [33]. As JavaScript applications are usually human-interactive (*e.g.*, visit websites), the security of JavaScript engines is needed to be taken seriously. The adversaries may hack a computer via inducing the user to access a prepared web page containing malicious JavaScript, which exploits the vulnerabilities of the JavaScript engine [17].

Existing Approaches. Fuzzing has been demonstrated to be one of the most effective techniques for detecting vulnerabilities [30, 31, 50] in many large-scale real-world projects. AFL [1], as a representative mutation-based fuzzing technique, has discovered thousands of vulnerabilities. And many variants of AFL (*e.g.*, AFLFast [21], AFLGo [20] and AFLNet [46]) are also developed to enhance AFL. Although AFL and its variants have been demonstrated effective in some programs, they still perform poorly on testing JavaScript engines.

Specifically, the simple mutations (*e.g.*, bitflip, byteflip) in AFL are grammar-blind and usually generate invalid inputs that cannot pass the syntax parsing. Consequently, the generated test cases

* Xiaofei Xie and Feng Li are the corresponding authors.

only explore the shallow part of JavaScript engines (*i.e.*, the parser) while the core functionalities are less tested.

Some grammar-aware fuzzing techniques are proposed to generate syntactically correct inputs. Skyfire [53] and TreeFuzz [45] generate syntactically valid inputs via learning probabilistic models from a large corpus of inputs. jsfunfuzz [9] generates JavaScript inputs based on the manually written grammar rules. Differently, LangFuzz [34] and Superion [54] are mutation-based fuzzers, which convert the JavaScript inputs to Abstract Syntax Trees (AST) and perform the random combinations between different fragments extracted from ASTs (*e.g.*, sub-tree replacement). Such approaches can generate syntactically valid JavaScript inputs but the semantics are usually incorrect, causing runtime errors. For example, after the AST sub-tree replacement, an undefined variable may be used in the new statements.

To mitigate this challenge, some techniques such as CodeAlchemist [33] and DIE [44] try to generate semantically valid JavaScript inputs. Specifically, CodeAlchemist [33] performs the type analysis and def-use analysis to check whether two fragments can be combined, *e.g.*, whether variables are defined or types are matched. DIE proposes an aspect-preserving mutation for keeping the structures and maintaining the types of the original inputs such that the existing high-quality inputs that have explored deep parts of the JavaScript engine can be further used. However, although CodeAlchemist and DIE intend to be semantic-aware, the semantic validity is still low. Our evaluation results show that they can generate more than 40%+ semantically invalid JavaScript inputs.

Besides, existing grammar-aware and semantic-aware approaches mainly rely on the initial seed corpus for learning language model [53] or performing the mutation [33, 34]. The diversity of the seed corpus may limit the fuzzing capability. For example, if some methods of an object (*e.g.*, APIs) are never used in seed inputs, the existing fuzzers will not test the corresponding functionalities.

Compared with the large-scale engines (*e.g.*, CharkaCore, JavaScriptCore), the embedded JavaScript engines are usually trimmed and have some different syntax. Moreover, few public documents for these differences make it hard to construct the specific rules.

Challenges. In summary, there are two main challenges for testing the JavaScript engines effectively. ❶ **Validity**: the first challenge is how to ensure the correctness of not only syntax but also the semantics of the generated JavaScript inputs during fuzzing such that deep components could be explored. ❷ **Diversity**: the second challenge is how to improve the diversity of the generated JavaScript inputs such that more functionalities of the JavaScript engine could be tested. In addition, we expect to build a general fuzzer that can test different JavaScript engines without customized configuration.

The Proposed Approach. To address these challenges, in this paper, we propose a reflection-augmented JavaScript engine fuzzer named SoFi[1]. To generate *valid* test cases that are syntactically and semantically correct, SoFi firstly performs a fine-grained program analysis to 1) identify available variables with a path-sensitive analysis, 2) identify types of variables that are used with dynamic analysis and 3) identify types of variables that are not executed based on static analysis.

Although the fine-grained analysis can improve semantic validity, some invalid inputs could still be generated by the mutation. Some unsupported statements or functions may also be introduced during the mutation. We further propose a complementary strategy to repair the invalid mutants based on the error information during the dynamic execution. The fine-grained analysis and the automatic repair make SoFi generate more valid inputs. To improve the *diversity* of the generated JavaScript inputs, we propose a novel method that makes use of a dynamic reflection mechanism to identify new *attributes* and *methods* of an object, which are not seen in the seed corpus. The fine-grained analysis, automatic repair and reflection do not depend on the target engines, which makes SoFi more *general* to test different engines.

To evaluate the effectiveness of our approach, we implemented SoFi and evaluated it on 9 different JavaScript engines, including 4 large-scale engines (*i.e.*, ChakraCore, V8, SpiderMonkey and JavaScriptCore). We compared SoFi with 3 state-of-the-art JavaScript fuzzers: the grammar-aware fuzzer Superion [54] and two semantic-aware fuzzers CodeAlchemist [33] and DIE [44]. The experimental results demonstrated the effectiveness of SoFi than the baselines: 1) SoFi can generate high-quality test cases than the baseline fuzzers (2x fewer errors), 2) SoFi is more effective in finding new bugs and improving code coverage. In total, SoFi discovered 51 unknown bugs. We also conducted an ablation study to evaluate the effectiveness of guaranteeing *validity* and *diversity*. The results show that both of these properties are useful in discovering bugs and increasing code coverage.

Contributions. Our contributions are as follows:

- We propose a novel semantic-aware fuzzing technique for JavaScript engines based on a more fine-grained analysis, the automatic repair and the reflection. To the best of our knowledge, this is the first work that makes use of reflection to enhance the effectiveness of fuzzing.
- We develop SoFi and conduct an extensive study to evaluate the effectiveness of SoFi.
- We have discovered 51 new bugs in real-world JavaScript engines and 28 of them have been confirmed or fixed by developers.

2 BACKGROUND

2.1 JavaScript Dynamic Reflection

Reflection is the program's ability to manipulate variables, properties, and methods of objects at runtime. With reflection, we can obtain the methods and properties of an object at runtime. As shown in Figure 1, we can iterate the object through the `for` statement to get the properties and methods supported by the object `obj`. The operator `typeof` can be used to distinguish whether a property is an attribute or a method. If it is a method, we can add one statement to call it (Line 3). If it is an attribute, we can use the attribute in operations.

The reflection mechanism provides a way to identify available methods and attributes of objects, which are not used in the existing seed corpus. Specifically, the method `getOwnPropertyNames`

[1]SoFi is an abbreviation of "Survival of the Fittest", meaning that our approach aims to produce more fitting seeds (*i.e.*, Seeds of the Fittest).

```
for (var p in obj) {
  if (typeof(obj[p]) == "function") {
    obj[p]();
  } else {
    console.log(obj[p]);
  }
}
```

Figure 1: Example of dynamic JavaScript reflection

```
var x = null;
if (Math.random() > 0.5) {
  x = [];
} else {
  x = "hello";
}
```

Figure 2: Example of dynamic types

can get all enumerable and non-enumerable properties in an object, which can be used in further mutation. It is more automatic and accurate than manually writing rules. For example, with the reflection, we can insert a new statement for assigning values to attributes of `obj` or calling the methods of `obj`, which are not used in the initial seeds.

2.2 Dynamic Types of JavaScript

JavaScript is a just-in-time compiled programming language with dynamic types. The type of a variable can be different in different paths. Moreover, the type can also be changed at runtime.

As shown in Figure 2, the type of `x` is different in the `if` and `else` branches. Specifically, in the `if` branch, `x` is an `Array` type variable. In the `else` branch, `x` is a `String` type variable. Type matching should be considered during the fragment combination of fuzzing (e.g., the variable replacement). Otherwise, a type error will be thrown if the actual type of a variable is different from the expected type.

3 MOTIVATION

Figure 3 shows a motivating example, where a seed JavaScript test case (on the left side) is selected to generate new test inputs via mutations. Here we summarize the main challenges for generating diverse and semantically meaningful seed inputs.

Diversity. Existing JavaScript fuzzers [33, 34, 44, 54] generate new test cases by random combinations of fragments in the seed corpus (*e.g.*, inserting or deleting new statements). Thus, the diversity of generated test cases depends on the seed corpus. However, if some methods of an object are not included in the seed corpus or are rarely called, they can not be tested. For example, assume the `getUTCSeconds` function does not appear in any seed JavaScript file from the corpus. With this assumption, existing approaches could not add line M1 in Figure 3 during the mutation since they have no awareness of the existence of such a function. Even supposing `getUTCSeconds` is included in the seed corpus, existing approaches are hard to generate the statement (*i.e.*, `data.getUTCSeconds()`)

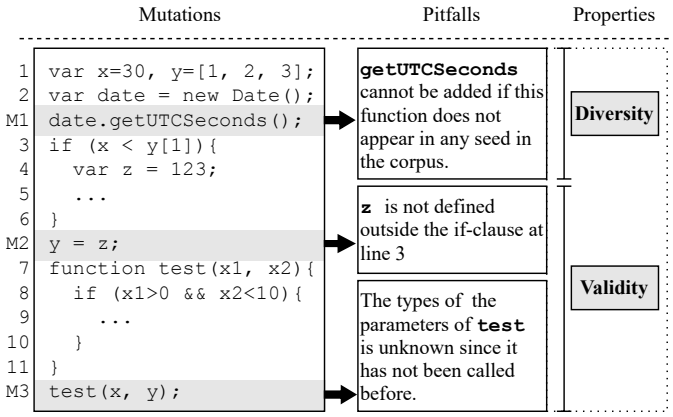

Figure 3: A motivating example. The left part shows the original seed input and the mutations applied on it. The middle part illustrates pitfalls introduced by the mutation and the right part shows the desired properties.

correctly due to that ① the probability of selecting the method `getUTCSeconds` from a large corpus of fragments is low and ② even if the method is selected, it is hard to match the correct object that belongs to the type by the random splicing [34]. To mitigate this problem, some researchers propose to use manually written rules to include methods for built-in objects [33, 44]. However, manually preparing such rules is time-consuming and labor-intensive.

Validity. Although some existing approaches can ensure the syntactic correctness of the generated test inputs [34, 54], they cannot guarantee the semantic validity of some statements. To increase the chances of semantic correctness, recent researches [33, 44] propose to conduct type inference for variables and function parameters. However, their type inferences still lack accuracy: ① the current analysis of variables is path-insensitive. For example, the variable `z` is defined in the if branch which is infeasible (line 4 in Figure 3). However, the path-insensitive analysis is coarse and `z` can be used in the new added statement (line M2 in Figure 3) during the mutation. ② the current analysis of variable/method parameter types relies on the `typeof` operator. However, if the method or variable is not used during the execution, their corresponding types cannot be inferred with `typeof`. For example, in Figure 3, the function `test` is not used in the original seed input, so its parameter types cannot be inferred and type mismatch can happen during the mutation (line M3). Moreover, existing approaches lack the ability of automatically repairing the generated test inputs.

In summary, the aforementioned example shows the main challenges in JavaScript engine fuzzing: ① the generation of test inputs relies on the initial seed corpus or manually written rules, which can limit diversity. ② there are many factors that may affect the semantic validity of generated inputs such as the unused variables and names, path-dependent types of a variable and the different features between multiple JavaScript engines. Motivated by the observations, this paper proposes a novel fuzzing technique SoFi which incorporates the reflection mechanism and advanced type

inference to generate more diverse and semantically meaningful test inputs.

4 APPROACH

In this section, we present a detailed description of SoFi. SoFi is a semantic-aware fuzzer, which is designed to generate *diverse* and *semantically valid* test cases such that SoFi can explore *broad* and *deep* implementation logic of the target JavaScript engine.

Overview. Figure 4 shows the overall design and workflow of SoFi. SoFi is implemented based on AFL [1]. The main contributions are highlighted with grey color while the other processes (*i.e.*, seed selection and coverage analysis) adopt the default strategy in AFL. To ensure semantic correctness, SoFi firstly performs an analysis on each JavaScript input to infer types of used/unused variables and identify unseen attributes/methods of objects with the dynamic reflection. At each iteration, SoFi selects one seed input from the processed seed corpus. Based on the selected seed input, the semantic-aware mutation and the reflection-based mutation are then performed to generate semantically *valid* and *diverse* inputs. Newly generated inputs are evaluated on the instrumented JavaScript engine. SoFi checks whether crashes are triggered and whether the JavaScript input causes some syntax/semantic errors. If there are some errors during the execution of the test input, an automatic repair strategy is adopted to fix these errors. If there are no crashes and the coverage is increased, the newly generated input is stored as a new seed. We describe each component as follows.

Pre-processing. Pre-processing is a fine-grained analysis that consists of three parts: 1) identify available variables at different positions with a path-sensitive analysis, 2) infer types of variables with static analysis and dynamic analysis and 3) identify attributes and methods of objects by dynamic reflection. If some types are still unknown after the static and dynamic type analysis, SoFi retrieves the similar fragments, whose types are known, from the historical data.

Mutations. Based on the pre-processing results, SoFi performs two types of mutations to generate new test cases: 1) mutate the current input by changing the constant value or expressions and 2) insert new statements to the current input, *e.g.*, unseen methods/attributes via reflection, statements from other inputs or unused function calls.

Execution. After new test cases are generated, they will be executed on the target JavaScript engine. If the runtime error is thrown with an input, other functionalities (after the erroneous statement) cannot be executed. The automatic repair is used to fix errors.

Example. We explain how SoFi addresses challenges in Figure 3. For M1, SoFi uses the runtime reflection to identify the possible variables and methods of the object `Date`. Even if `getUTCSeconds` is not included in the initial seed corpus, the runtime reflection can still identify this function. The reflection could also be used to check whether a given method/attribute is valid for the `Date` type. If not, the invalid method/attribute will not be used during the mutation. For the variable z in M2, path-sensitive analysis can identify the available variables at each position. Specifically, SoFi infers that `z` cannot be used at line M2 since the if branch is not executed. For M3, although `test` is not called in the original seed, our static type inference can still analyze types of its parameters $x1$ and $x2$. Suppose the static analysis cannot identify types and `test` is called in the other seed inputs. In that case, SoFi retrieves types from the historical knowledge (*i.e.*, the type information from previous seed inputs).

4.1 Pre-processing

Recall that our objective is to make SoFi satisfy the two properties, *i.e.*, *diversity* and *validity*. Therefore, we firstly perform a fine-grained pre-processing combining dynamic analysis and static analysis on each seed input for identifying available variables, inferring types of variables and finding unseen methods and attributes of an object, which will be used in the mutation stage.

4.1.1 Path-Sensitive Variable Identification. Identifying available variables is vital for the semantic correctness of generated inputs. Using unavailable variables will throw the undefined variable error. Existing seed generation methods either do not consider the availability of variables or adopt a coarse path-insensitive analysis that cannot recognize variables defined in the un-executable branches. To precisely capture the available variables, we perform a path-sensitive analysis to identify variables.

Figure 5 shows the instrumentation for identifying available variables after each statement. Specifically, we first traverse the AST of the given seed and get all defined variables (Line 4). Then we instrument the code snippet (Line 5-15) after each statement (suppose its line number is `loc`). After each statement, we access the variables with the `try-catch` statement. If a variable is accessed successfully (Line 8), the variable is available at this location. Finally, all available variables at line `loc` are saved in `liveVar` (Line 16).

4.1.2 Type Inference. As JavaScript is a dynamic typed language, we need to analyze types of variables for semantic-aware fuzzing. We propose a series of type inference techniques based on dynamic and static analysis to infer the types more accurately. The dynamic analysis could obtain accurate types of variables that are used in the dynamic execution. Static analysis is adopted to analyze types of variables, which are not used in the execution but may be used after the mutation. If types cannot be inferred with the static analysis, we use a historical knowledge-based strategy to identify similar code fragments whose types are known in previous mutations.

Dynamic Type Inference. Existing approaches (*e.g.*, [33]) mainly use the operator `typeof` to perform dynamic type analysis, which may lead to inaccurate inferences. For instance, `Array` is recognized as an `Object` using `typeof` analysis, which is too coarse. `Array` can be matched with other objects (*e.g.*, `Date`) that appeared in the seed corpus during mutation.

SoFi dynamically infers the type of an object e with the rules in Figure 6. Specifically, SoFi adopts these two rules to check the type of each available variable (at Line 10 of Figure 5). It first identifies the type of an object through the name of the object's constructor (Rule *T1*), which is much more accurate than `typeof`. For example, by the constructor name, we can get the `Array` type. If it fails, then the `typeof` operator will be used.

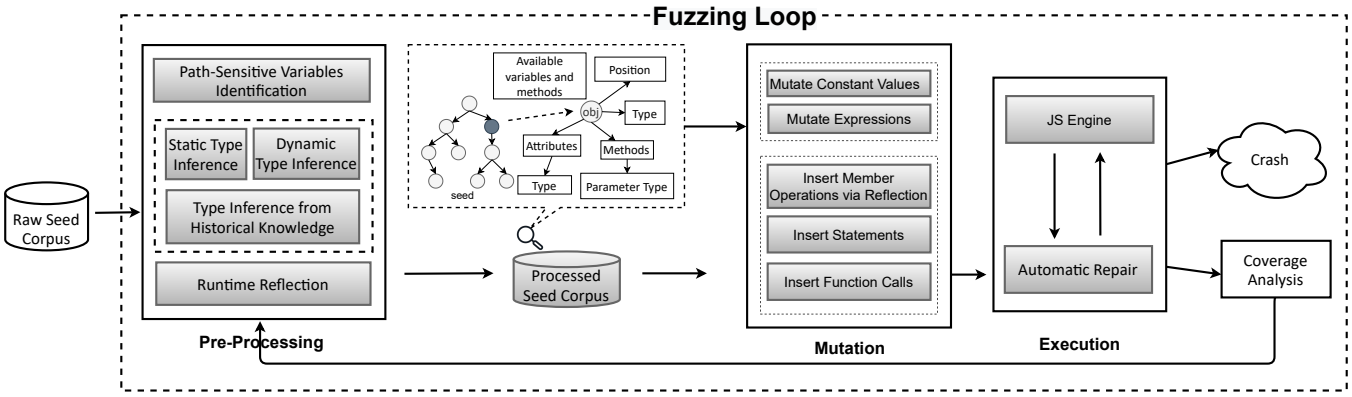

Figure 4: The Overview of SoFi

```
1  //loc is the current line number.
2  //liveVar is a dictionary saving available
3  //variables after each statement.
4  var varSet = getAllVars();
5  for (value in varSet) {
6    var tmpSet = new Set();
7    try{
8      if (value != undefied) {
9        tmpSet.add(value);
10       ...
11     }
12   } catch(err) {
13     //unavailable var
14   }
15 }
16 liveVar[loc] = tmpSet;
```

Figure 5: Instrumentation for identifying available variables

$$T1(e) = eval(e.constructor.name)$$
$$T2(e) = typeof(e)$$

Figure 6: Dynamic Type Inference Rules

Static Type Inference. Some codes can not be executed in a program. For example, some defined variables or declared functions that are not executed under the current context. However, they may be executed after the mutation changes the context (*i.e.*, the precondition). Existing approaches fail to analyze types for such variables, which may be used after the precondition is changed during the mutation. We propose a static approach to analyze types of variables and parameters of functions, which are not used in the dynamic execution of the current input.

Specifically, we first parse the JavaScript seed into an AST tree, and then traverse the AST and determine types of variables with a set of defined rules (see details in Figure 7). *T3* uses the transfer characteristics of the assignment expression since types of the *lvalue* and the *rvalue* should have the same type. Rules *T4-T8* infer the types based on the expression type of AST nodes. For instance, the variable is a Function if it is a `FunctionExpression` or a Boolean type if it is a `LogicalExpression`.

However, some variables that cannot be analyzed from the expression type. For example, `c = a * b` is an assignment expression in the AST, where `a * b` is a binary expression, which cannot be determined by Rules *T4-T8*. From the operator `*`, we can infer that `a`, `b` and `c` are `Number` types. Based on the observation (Rules *T9* to *T13*), we define some rules to identify types with the operator in the expression. Specifically, if the expression is to create a new object (*T9*), then we can obtain the type of the new object based on its constructor function name. In addition, we determine the type based on the related operator (denoted as *e.operator*) on the variable *e* (*T10-T13*). For example, if the operator is $*$ and /, *e* is a `Number` object. If the operator is &&, *e* is a `Boolean` object. If the operator is *e.substr*(), *e* is a `String` object. For complex expressions, such as member variable expressions or other complex structures, we recursively disassemble them into simple expressions based on their structures and infer their types with the defined rules.

Some expressions cannot be inferred, such as `c = a + b`. Both `Number` and `String` support `+` in JavaScript. In this case, we reduce the candidate types to `Number` or `String` and continue to analyze based on other operations on `c`. For instance, if *c* is used in another statement `d = c * a`, then we can infer that `a` and `b` are `Number` types. If *c* is used in `d = c.spilt("")`, then `a` and `b` are likely to be `String` type.

Consider an example shown in Figure 8, where `hello` is not called. Hence, dynamic analysis cannot obtain types of parameters in `hello`. Our static analysis can be used to analyze types of `ttt` and `pp` based on the relevant operations. Using the static rules, we can infer that they are `Number` and `String`, respectively.

Historical Knowledge based Type Inference. If both dynamic and static analysis cannot infer types for variables in a seed input, we propose to infer types based on the knowledge of the historical data. For example, suppose the types of parameters in the method `toFixed` is unknown in one seed. If there is another seed (in the seed corpus) that calls the method `toFixed`, which has been analyzed with dynamic type analysis, then we can retrieve the types based on the previous knowledge. Specifically, during the static/dynamic type

$T3(e.left) = T(e.right)\ \{if\ e\ is\ assignExpression\}$

$T4(e) = Function\ \{if\ e\ is\ FunctionExpression\}$

$T5(e) = Boolean\ \{if\ e\ is\ LogicalExpression\}$

$T6(e) = Number\ \{if\ e\ is\ ArithmExpression\}$

$T7(e) = String\ \{if\ e\ is\ StringExpression\}$

$T8(e) = Array\ \{if\ e\ is\ ArrayExpression\}$

$T9(e) = right.callee\ .name\ \{if\ e\ is\ NewExpression\}$

$T10(e) = Boolean\ \{if\ e.opeator\ in\ LogicalOperatorSet\}$

$T11(e) = Number\ \{if\ e.opeator\ in\ ArithmeticOperatorSet\}$

$T12(e) = String\ \{if\ e.opeator\ in\ StringOperatorSet\}$

$T13(e) = Array\ \{if\ e.opeator\ in\ ArrayOperatorSet\}$

Figure 7: Static Type Inference Rules

```
var bb = 1000000, ba = 'mmm';
function hello (ttt, pp) {
  ttt = ttt * 5;
  pp = pp + 'Tom';
  console.log('hello world!!!', ttt, pp);
}
```

Figure 8: Example of static type inference

analysis, we record all inferred types of variables and parameters in methods, which are used for the type retrieval in the future.

4.1.3 Unseen Methods and Attributes Identification. To further identify methods or attributes of an object, which are never used or rarely used in the seed corpus, we provide a reflection-based analysis to determine each object's available attributes and methods during the dynamic execution. We instrument the reflection code for each available variable (at Line 10 of Figure 5) to collect attributes and methods of the object, which will be used in the mutation stage.

Specifically, in JavaScript, there are enumerable or non-enumerable properties, which can be obtained through `getOwnPropertyNames`. In addition, some methods and properties are defined in the `prototype`. In order to get complete properties, we also need to analyze the `prototype` of the object through the method `getPrototypeOf`. Figure 9 shows the instrumented code for each variable *obj* with the reflection method `getOwnPropertyNames` and `getPrototypeOf`. `getOwnPropertyNames` is used to analyze the common properties (Line 6-13) and `getPrototypeOf` is used to analyze the `prototype` of the object (Line 14-23). Finally, the identified methods and attributes will be stored for further mutation.

4.2 Semantic-aware Mutation

After a seed input is selected, SoFi performs a semantic-aware mutation to generate semantically valid inputs. Specifically, SoFi mainly contains two types of mutations, *i.e.*, mutate the existing elements of the current input and insert new elements into the input. Note that we do not introduce the deletion operator due to that random deletion is more likely to affect the semantic validity.

```
1  // obj is an available variable
2  // attrs and methods store the identified
3  // attributes and methods for the variable obj
4  var attributes = new Set();
5  var methods = new Set();
6  Object.getOwnPropertyNames(obj).forEach(
7    (name)=>{
8      if(typeof obj[name] =='function') {
9        methods.add(name);
10     } else {
11       attributes.add(name);
12     }
13 });
14 var proto = Object.getPrototypeOf(obj);
15 Object.getOwnPropertyNames(
16   proto.forEach((name) => {
17     if (typeof obj[name] == 'function') {
18       methods.add(name);
19     } else{
20       attributes.add(name);
21     }
22   }
23 );
```

Figure 9: Instrumentation of dynamic reflection

Algorithm 1: SemanticMutate

Input: *p_ast*: the AST of the processed input
Output: A mutated input

```
1  type ← pickMutationType();
2  if type == mutation then
3      mtype ← selectMutation();
4      if mtype == leaf then
5          consNode ← getConsLiteralNode(p_ast);
6          changeToSpecialVal(p_ast, consNod);
7      else if mtype == expression then
8          exp ← selectExp(p_ast);
9          new_exp ← buildNewExp(exp);
10         replaceExp (p_ast, exp, new_exp);
11 else if type == insertion then
12     itype ← selectInsertion();
13     if itype == reflection then
14         obj, loc ← randObjSelect(p_ast);
15         property ← getRareMethods(obj);
16         insertPropChange(obj, property, loc);
17     else if itype == statement then
18         stmt ← selectStmtFromOtherSeed();
19         randomInsert(p_ast, stmt);
20     else if itype == fuction then
21         function ← selectAvalibleFunction();
22         insertFunctionCall(function);
```

Algorithm 1 shows the detailed mutation on a given AST of the processed input. SoFi randomly selects the mutation operators, *i.e.*, whether *mutate* existing elements or *add* new elements (Line 1).

Mutate Existing Elements. SoFi adopts two mutation operators, *i.e.*, to mutate the constant values or the expression:

- *Mutate Constant Values.* Some vulnerabilities are often caused by special values (*e.g.*, the boundary values), SoFi detects all constant values in the leaf nodes of *p_ast* and randomly select one node (Line 5). Then the value is changed by selecting a random value or assigning a special value, *e.g.*, `Number.MIN_VALUE`, `Number.MAX_VALUE` (Line 6).
- *Mutate an expression.* SoFi randomly selects one expression *exp* (Line 8), *i.e.* a sub-AST in *p_ast*. Based on the type information of *exp*, a new expression *new_exp* is created (Line 9). Specifically, the new expression can be built by: creating a constant value, randomly selecting a type-matched expression, randomly selecting a type-matched variable or an attribute of an object or randomly creating a function call that returns the matched type of *exp*, *etc.* Note that the selected variable or function must be available in the current context. Finally, SoFi replaces the sub-AST *exp* with the new one *new_exp* (Line 10).

Insert New Elements. To further improve the diversity, three different *insertion* mutations are proposed:

- *Insert member operation via reflection.* SoFi first randomly selects an available object *obj* at the position *loc* (Line 14). Based on the pre-processing analysis results (see Section 4.1.3), SoFi identifies methods and attributes that are unseen or rarely used in the seed corpus (Line 15). Then we can insert a statement that changes the value of the attributes or calls the selected methods (Line 16). Specifically, the value of an attribute can be assigned with a constant value or a type-matched expression. For the method of an object, if the method call exists in previous seeds, we can retrieve the types from the historical knowledge. If it is unseen, we construct the parameters with an empirical strategy by enumerating parameters or not using parameters.
- *Insert a statement from other seed inputs.* SoFi inserts the statement selected from other inputs into the current input (Line 18-19). Note that the new inserted statement (*e.g.*, loop statement, if statement) may have a large impact on the semantic correctness of the seed input. We adopt a more conservative way to select statements that have less impact on other code, *i.e.*, the semantics of the original test case is unlikely to change after the insertion. For example, we can insert the variable definition, the class definition, the function definition or the statement with basic operators (*e.g.*, arithmetic operators, string operators). Such variables or functions never appeared in the JavaScript programs. Thus they will not affect the context at the insertion location.
- *Insert a Function Call.* During the mutation, some declared functions may not be executed. SoFi selects the available functions that are defined in the current input (Line 21) and creates a function call at a position (Line 22).

4.3 Heuristic-based Repair

After the mutation, some semantic errors may still occur. For example, the type inference is not accurate or the JavaScript engine does not support some syntax. Such errors can reduce the effectiveness of fuzzing because once one of the statements causes an error, the following code will not be executed. To mitigate this challenge, we propose a heuristic-based technique to fix errors in the input.

```
1 ReferenceError: 'foo' is not defined
2   at Global code (./queue/ReferenceError.js:1:1)
```

Figure 10: An example of an error message

We perform a study on the invalid mutants and summarize the common errors (*e.g.*, reference error, type error, range error, URI error). Based on that, we propose a rule-based approach to fix these errors. Specifically, we localize the error to identify the root cause of the error and erroneous elements. Then, we adopt specific rules to repair different errors. If errors cannot be fixed, we try to delete the erroneous statements.

```
1 abs_seedname + ":(\d+)"
2 seedname + ":(\d+)",
3 "Error: Line (\d+)"
```

Listing 1: Identify Lines

4.3.1 Error Localization. The localization is based on error messages from the JavaScript engine. We apply the regular expressions to parse the error message such that the line number and the erroneous elements can be identified. For example, Figure 10 shows an example of the error message, and we identify the line number (*i.e.*, line 1 in `ReferenceError.js`) and the erroneous element (*i.e.*, the variable `foo`), which will be used in the repair. In Listing 1 and Listing 2, we show two examples of regular expressions which can parse the error message.

```
1 "SyntaxError: Illegal\s+(\w+)\—expression"
2 "RangeError:\s+(\w+)\(\)\s+argument"
```

Listing 2: Identify Erroneous Elements

4.3.2 Rule-based Repair. We design rules for fixing common errors. Note that more rules can be added, which is left as our future work. After the line number and the erroneous elements are identified, we traverse the AST of the input to match the corresponding AST nodes and fix the errors as follows:

- **ReferenceError.** For the reference error, we identify the variable that is not defined and insert a statement to declare it before the variable reference.
- **TypeError.** For the type error, if we can identify the required type from the error message, then a new definition statement that defines the variable with the matched type is inserted to fix the error. Otherwise, we will enumerate variables with common types (*e.g.*, String, Number, Boolean and Object) until the error disappears.
- **RangeError.** For the range error, based on identified erroneous elements, we adjust the range such that it is in the legal range. For example, in the array indexing, a large index may cause out of bounds error. We decrease the index value until it is in a reasonable range.

- **URIError.** A common error is that the URI can be broken during the random mutation. For such errors, we will generate a valid URI to replace it.

For errors that cannot be handled with the aforementioned rules, we adopt a remedy strategy by deleting the erroneous code snippet. Specifically, the deletion is started from erroneous statements that are identified from the error message. However, some deletions may cause errors in other statements. In this case, we continue the repair process by applying rule-based repair or the deletion strategy.

5 EVALUATION

We have implemented SoFi with JavaScript, C and Python [2]. As shown in Figure 4, SoFi is a variant of AFL, in which we added the seed pre-processing module, replaced the original grammar-blind mutation with the semantic-aware mutation and implemented an automatic repair strategy in the evaluation part. To improve the analysis efficiency, the pre-processing results are cached for the semantic-aware mutation. In detail, we use Esprima [6] to parse the JavaScript code and get the AST. ast-type [2] is applied to traverse the AST for further analysis. After the AST is mutated, we use escodegen [7] to generate the JavaScript code from the AST.

5.1 Experimental Setup

To evaluate the effectiveness of SoFi, the experiments are designed to answer the following research questions:

- **RQ1:** Can SoFi outperform the state-of-the-art JavaScript fuzzers in terms of finding new bugs and improving code coverage?
- **RQ2:** How do the 2 properties (*i.e.*, diversity and validity) impact the performance of SoFi?
- **RQ3:** How effective is SoFi in generating semantically valid inputs?
- **RQ4:** Can SoFi find new bugs in real-world JavaScript engines?

5.1.1 Benchmarks. We selected 9 widely used JavaScript engines to evaluate our approach. Specifically, we selected 4 large-scale JavaScript engines, *i.e.*, ChakraCore (1.12.0.0), V8 (9.0.0), JavaScriptCore (2.24.4) and SpiderMonkey (C72.0a1), which are used in popular web browsers (*i.e.*, Microsoft Edge, Safari and Mozilla). Since these engines have been tested by the baselines, the bugs found earlier by baseline fuzzers may have already been fixed. This could affect the comparison of bug finding. Therefore, we selected another 6 JavaScript engines which have not been tested by the baselines before, *i.e.*, Moddable (10.3.2), MuJS (1.0.7), Jsish (3.0.4), Jerryscript (2.2.0), Quickjs (2020-07-05). These 6 engines are also widely used in many scenarios, *e.g.*, IoT devices, micro-controllers, PDF readers, mobile applications, etc. In summary, we selected the diverse and representative JavaScript engines for evaluating the effectiveness of SoFi.

[2]The source code can be found on the website[13].

5.1.2 Baselines. We select 3 JavaScript fuzzing approaches as the baselines to evaluate the effectiveness of SoFi. Specifically, we select a state-of-the-art grammar-aware JavaScript fuzzer (*i.e.*, Superion [54]) and 2 state-of-the-art semantic-aware JavaScript fuzzers (*i.e.*, CodeAlchemist [33] and DIE [44]).

Note that CodeAlchemist and DIE require the manual configurations for the target JavaScript engines (*i.e.*, the syntax information). By default, their public versions only include the configuration for some JavaScript engines. We tried our best to configure the syntax for other unsupported engines. However, it requires a lot of work, including changing source code and strict grammar matching to support other JavaScript engines, making the extension quite difficult. Instead, we use the configuration of ChakraCore to conduct the fuzzing for other unsupported engines, which can still guarantee the syntax and semantic correctness in most cases.

5.1.3 Setup. We totally selected 40,774 JavaScript files as the initial seed corpus including: 1) seeds from the official ECMAScript conformance test suite *Test262* [14], js-vuln-db [15] and 2) seeds that are crawled from a large number of web pages by us. The corpus has been trimmed by AFL's `cmin` tool and deduplicated.

To answer the aforementioned research questions, we designed a set of experiments. Specifically, to answer **RQ1**, each fuzzer was used to test selected JavaScript engines given a time budget of 24 hours. To be fair, all tools are initialized with the same seed corpus js-vuln-db [15]. We do not use all 40k+ seed inputs because CodeAlchemist and DIE take a long time to process them. To evaluate the coverage, we follow the same settings in FairFuzz [38] and Evaluate-Fuzz-Testing [36] to measure the edge coverage of test cases, which is calculated based on AFL's bitmap coverage. Notably, the *preprocessing* phase in SoFi is performed after a seed is selected for mutation. Thus, the preprocessing time is included in the fuzzing time of SoFi. It does not affect the fairness of the comparison. For the bug finding experiments, the target engines are compiled with AddressSanitizer such that more bugs can be captured. We compared SoFi with baselines on the edge coverage and the number of unique bugs. To mitigate the randomness, we repeat the process 10 times and calculate the average results.

To answer **RQ2**, we conducted an ablation study to evaluate the effectiveness of SoFi without diversity or validity.

To answer **RQ3**, *i.e.*, measure the validity of generated test cases, we slightly modified each fuzzer to save every test case even if this test case cannot increase the coverage (*i.e.*, without coverage feedback). Given the same initial seed corpus, we ran each fuzzer until 100,000 test cases are generated. Then we checked the validity of such test cases. A test case is valid if no runtime error is triggered when the test case is executed on the JavaScript engine. Note that the calculation of the error rate in our paper is different from Superion and DIE, where only the validity of seeds in the AFL queue are measured while the discarded mutants are not considered. Since all mutants will be executed during fuzzing and can significantly impact the fuzzing performance, we measured the validity for all mutants.

To answer **RQ4**, we ran SoFi to test each engine for 2 weeks and collect the unique bugs discovered. In this experiment, we used all 40k+ seed inputs.

5.2 RQ1: Comparison with Baselines

Bug Finding. Table 1 shows the average number of unique bugs found by each fuzzer in 24 hours. The bug classification is based on the AddressSanitizer results and the manual analysis. Since the baselines have already been applied to test ChakraCore, V8, JavaScriptCore and SpiderMonkey in previous works and some bugs detected on these targets earlier may have been fixed. It may not be fair to compare the number of bugs found on these 4 JavaScript engines. However, for the other 5 untested engines, we can see that SoFi can find more bugs.

Consider the results in the 5 JavaScript engines, we found that SoFi can detect new bugs on MuJS where all the other techniques cannot find any bug in 24 hours. Although the grammar-aware fuzzer Superion can execute faster, SoFi still discovers more new bugs on the 5 JavaScript engines. This is because Superion is semantic-blind and generates many invalid inputs.

For the semantic-aware fuzzers, we found that despite the semantic awareness, CodeAlchemist and DIE still generated many semantically invalid test inputs, which can affect their effectiveness.

The last column in Table 1 shows the unique bugs which are detected by SoFi but not by other baseline fuzzers. In total, there 9 bugs which can only be detected by SoFi.

Code Coverage. Figure 11 shows the average number of edges achieved on the 9 JavaScript engines by different fuzzers.

From Figure 11, we can clearly see that SoFi significantly outperforms other techniques on 7 out of the 9 tested JavaScript engines with the lower bound of its shaded area higher than the upper bound of the shaded area of any other techniques. However, on ChakraCore, JavaScriptCore and Moddable, we found that SoFi performed the same or worse than DIE. Interestingly, although SoFi covered fewer edges than DIE on Moddable and ChakraCore, it can detect more bugs on average. With further investigation, we found that the inputs that trigger 2 of the bugs detected by SoFi are generated via the reflection mutation and DIE lacks the specific rule to include the related object functions. Thus, these 2 bugs can never be detected by DIE. This explains why SoFi can detect more bugs while it may not have a coverage advantage and also demonstrates the importance of the diversity for the generated test inputs.

> **Answer to RQ1:** With the analysis of the results in Table 1 and Figure 11, we can positively answer RQ1 that SoFi significantly outperforms the state-of-the-art JavaScript engine fuzzing techniques in terms of coverage and detecting new bugs. In addition, SoFi can detect 9 unique bugs that cannot be found by other fuzzers.

5.3 RQ2: Impact of Diversity and Validity

We evaluate the impact of the design choices for diversity and validity with an ablation study. For this purpose, we modified SoFi to build SoFi _V and SoFi _D. ❶ In SoFi _V, we removed path-sensitive analysis, type inference as well as automatic fixing components. By comparing the results between SoFi _V and SoFi, we can demonstrate the importance of the validity of the test inputs because the removed components of SoFi _V are mainly related to ensuring the test input validity. ❷ In SoFi _D, we removed the reflection mutation that contributes to the diversity of the generated test inputs. By comparing the results between SoFi _D and SoFi, we can demonstrate the importance of input diversity.

Table 1 shows the average number of bugs detected by SoFi _V, SoFi _D and SoFi in 24 hours. We can see that without ensuring validity, SoFi detected around 8 fewer bugs on average, while without reflection-based mutation, SoFi detected about 5 fewer bugs. This shows that both validity and diversity are important for bug detection and validity is a possibility more important because it is more fundamental for covering deep logic in the JavaScript engines.

Figure 11 shows the number of edges covered by SoFi _V, SoFi _D and SoFi in 24 hours. We can see that similar to the results of the bug detection, both validity and diversity can help improve code coverage and having both of them together can yield the best results.

> **Answer to RQ2:** The results demonstrate that both validity and diversity are useful in SoFi. By integrating these 2 together, the performance of SoFi can be significantly improved.

5.4 RQ3: Validity of Generated Test Cases

We perform a deep analysis on the validity of test cases generated on ChakraCore. Figure 12 shows the detailed results of different fuzzers. To evaluate the usefulness of the automatic fix in improving the quality of test cases, we also evaluate the fuzzer SoFi (nofix) that removes the automatic fix module.

The results show that Superion generates much more erroneous inputs than others, where SyntaxError accounts for a large proportion. It is because Superion adopts the random sub-tree replacement operator, deletion and the AFL built-in mutation operators, which are more likely to break the syntax. Consider the semantic-aware fuzzers, without automatic repair, SoFi (nofix) could achieve similar results with CodeAlchemist and DIE. DIE is relatively better because it is more conservative to preserve the structure and types. However, SoFi (nofix) has a low error rate on the TypeError, which shows the effectiveness of our fine-grained type inference (see Section 4.1). The error rate of ReferenceError is higher since the expression-based mutation may introduce errors. After adding the automatic fix, the overall error rate is reduced and many ReferenceErrors are also reduced.

> **Answer to RQ3**: The results demonstrated that the fine-grained analysis and the automatic repair could improve the validity of the generated test cases.

5.5 RQ4: Results of Real-world Bugs

Table 2 summarizes the discovered bugs and the description, including a total of 51 bugs, where 10 CVEs have been assigned. The third column shows the description of discovered bugs and other columns list the JS engine, the ID of bugs detected and the status of the bugs, respectively.

Notably, we found 25 bugs on the large-scale engines, *i.e.*, 18, 5, 1, and 1 bug on ChakraCore, SpiderMonkey, JavaScriptCore and V8, respectively. The results demonstrated the effectiveness of SoFi in finding new bugs on real-world JavaScript engines. We further made an in-depth analysis on these bugs and found that reflection

Table 1: Average number bugs found by each fuzzer in 24 hours

JS Engine	SoFi _V	SoFi _D	SoFi	CodeAlchemist	DIE	Superion	Unique Bugs Discovered by SoFi
ChakraCore	0	0	0.4	0	0	0	1
V8	0	0	0	0	0	0	0
JavaScriptCore	0	0	0	0	0	0	0
SpiderMonkey	0	0	0	0	0	0	0
Moddable	1	3	5.4	1.4	1.8	2.6	2
MuJS	0	0	0.8	0	0	0	1
Jsish	2.5	4.2	5.2	0.2	1	0.4	4
Jerryscript	0	0	0	0	0	0	0
Quickjs	0	0	0.4	0	0.2	0	1
Total	3.5	7.2	12.2	1.6	3	3	9

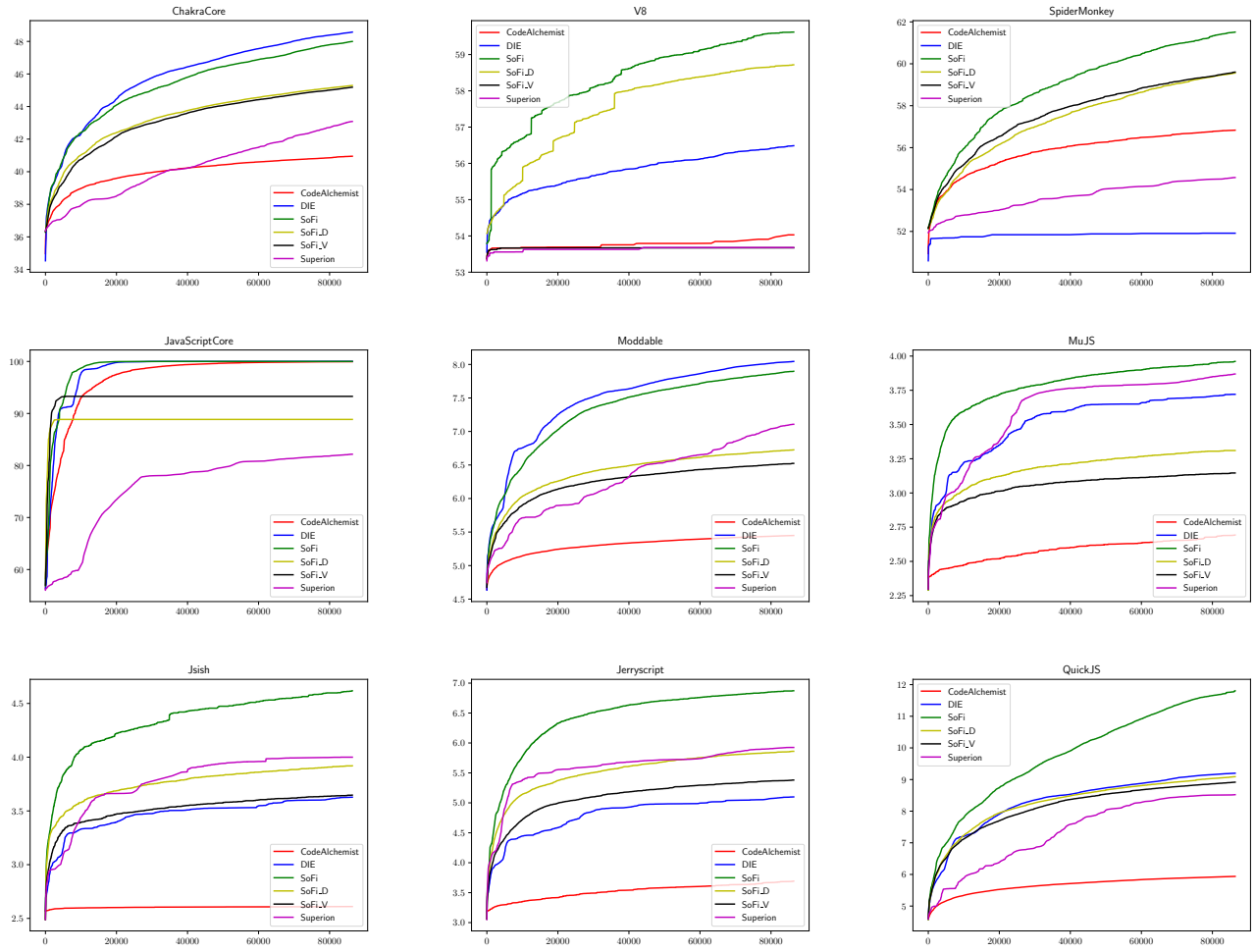

Figure 11: Number of edges covered over time for all the evaluated techniques. The y-axis is the number of edges; the x-axis is the time (in seconds).

- We run Mann-Whitney U Tests and the p-values are all <0.05, indicating statistical significance.

improves the effectiveness of SoFi a lot. We show 2 confirmed cases as follows:

Case1: Jsish stack overflow-*insert unseen method call of an object via reflection*. Figure 13 shows a test input generated by SoFi,

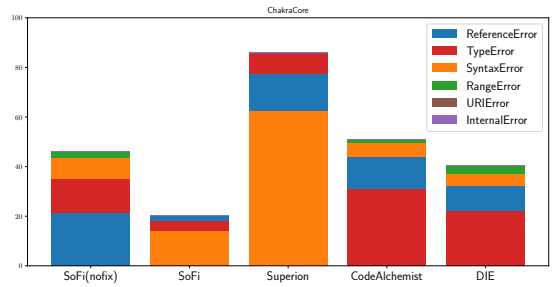

Figure 12: The error rate of test cases generated on ChakraCore

which triggered a stack overflow vulnerability on Jsish. SoFi inserts 2 statements at Line 2 and Line 3 during the semantic mutation. It is worth noting that `toPrecision` is not included in the initial seed corpus. The *reflection* mechanism could identify this method from the Number Object. Specifically, with the *insertion mutation*, we firstly add these two statements. Then with the *expression mutation*, `xx` is changed to `~xx`. The default value of `xx` is `0`, thus its inversion is `0x7fffffff`, which is too large and causes the stack overflow.

```
1 var num = 0;
2 var xx;  /*newly added*/
3 var AkBW = num.toPrecision(~xx); /*newly added*/
```

Figure 13: Stack overflow in Jsish

Case2: Jsish heap overflow-*insert the use of an attribute via reflection.* Figure 14 shows another test input generated by SoFi and it causes the heap overflow. With the reflection, a new statement that changes the attribute of *o* is generated. Specifically, the `length` is assigned with a signed negative number (*i.e.*, -9007199254740091) and is treated as a big unsigned integer later. The unsigned integer multiplied by a fixed value (*i.e.*, the unit size of the array) is used to apply a heap block. Due to the overflows caused by the multiplication, a very small heap is applied and subsequent access to the heap block will lead to a heap overflow.

Answer to RQ4: The results demonstrated that SoFi is useful in detecting unknown bugs in JavaScript engines, where reflection can significantly improve the performance of SoFi in terms of bug detection.

```
1 var o = [1, 2];
2 o.length = -9007199254740091; /*newly added*/
3 o.unshift(o.length);
```

Figure 14: Heap overflow in Jsish

Table 2: Summary of discovered vulnerabilities.

JS Engine	ID	Descriptions	Status
ChakraCore	1	incorrect abort in EntryIntl_PluralRulesSelect	Issue6633
	2	AssertOrFailFast in MapStFldHelper	Issue6683
	3	FatalInternalError	Issue6629
	4	non-fatal for length too big	Issue6627
	5	out-of-bounds error from string SetLength	Issue6625
	6	ASMModule is called with invalid params	Issue6624
	7	repeated expensive interactions with a large array	Issue6623
	8	Memory leak	Issue6642
	9	Segmentation fault	Issue6643
	10	Invalid memory read	Issue6644
	11	READ memory access	Issue6645
	12	Segmentation fault	Issue6646
	13	Segmentation fault	Issue6647
	14	Segmentation fault	Issue6648
	15	Segmentation fault	Issue6649
	16	Segmentation fault	Issue6651
	17	Memory leak	Issue6652
	18	Memory leak	Issue6654
SpiderMonkey	19	Over write	Bug1672679
	20	Over write	Bug1672683
	21	Over write	Bug1672678
	22	READ memory access	Bug1672677
	23	Assertion failure	Bug1708976
JavaScriptCore	24	Crash calling the " load" function	Bug222542
V8	25	Debug check failed while parsing number	Issue1188571
Jsish	26	Integer overflow	CVE-2020-22875
	27	Stack overflow	CVE-2020-22907
	28	Heap overflow	CVE-2020-22874
	29	Heap overflow	CVE-2020-22907
	30	Use after free	CVE-2020-27522
	31	Memory leak	Issue17
QuickJS	32	Stack overflow	Issue13
	33	Heap overflow	CVE-2020-22876
JerryScript	34	Reference error	Issue3674
MuJS	35	Buffer Overflow	CVE-2020-22885
	36	Buffer Overflow	CVE-2020-22886
Moddable	37	Type Confusion	CVE-2020-22882
	38	Use after free	CVE-2020-25465
	39	Heap overflow	Issue377
	40	NULL pointer dereference	Issue378
	41	NULL pointer dereference	Issue379
	42	NULL pointer dereference	Issue380
	43	Stack overflow	Issue364
	44	Stack overflow	Issue376
	45	Heap overflow	Issue580
	46	Heap overflow	Issue581
	47	Heap overflow	Issue582
	48	Heap overflow	Issue583
	49	Over access	Issue585
	50	Stack overflow	Issue586
	51	Stack overflow	Issue587

6 THREAT TO VALIDITY AND DISCUSSION

6.1 Threat to Validity

The main external threat is the selected benchmarks, which may cause bias for the comparative results. To address this issue, we have

selected a wide range of real-world JavaScript engines. To the best of our knowledge, we have used the largest number of JavaScript engines for evaluation among the recent related works, including 4 widely used engines and 5 other engines. Another external threat is that the used seed inputs are also a threat to the validity of the results. We collected a large number of JavaScript files that are used by each fuzzer.

6.2 Discussion

We discuss the two properties of SoFi as follows:

- **Validity**. Theoretically, achieving high validity of the test cases is not necessary for detecting bugs in JavaScript engines. In fact, there is a balance between the validity guarantee and bug detection. High-validity test cases may miss some bugs that are caused by the JavaScript parser and require more time overhead. On the other hand, low-validity test cases may waste lots of testing time because most of them can be filtered by syntax/semantic checking, making the deeper functionalities unexplorable. Consider that many shallow bugs caused by the parser have been discovered. Hence, in SoFi, we prefer to spend more effort in guaranteeing the validity improvement to detect deep bugs.
- **Diversity**. Compared with the validity, improving the diversity of test cases can clearly enhance the capability of fuzzers by generating new test cases. It is worth emphasizing that, with reflection, SoFi can discover bugs that others *cannot* discover (e.g., DIE, CodeAIchemist) since reflection can provide unseen methods or attributes. The qualitative analysis (see Figure 13 and Figure 14) has shown the new bugs that can only be detected by SoFi.

There are still some limitations that need to be improved:

- Historical Knowledge-Based Type Inference may miss some type information if types are not correct in the history or not included in the history; One potential solution is to introduce the machine-learning technique to infer types.
- The repair is a rule-based approach, and the rules are not complete. How to better repair the test case can be one future work.
- The parameters of the functions identified by the reflection may be unknown. A future work is to mine the specification from the JavaScript documents, which could better complement the reflection-based method.

7 RELATED WORK

7.1 General-purpose Fuzzing

Fuzz testing is a practical method to detect software vulnerabilities. It tests software by randomly generating the malformed test cases.

Mutation-Based Methods. The mutation-based methods generate new test cases by mutating the files directly in the seed corpus. AFL [1] is one of the most famous tools in this direction, which implements multiple conventional mutation methods with the coverage-guided framework. Following AFL, there are many AFL variants [21, 23, 25–27, 38–40, 52, 56] that are proposed to enhance the effectiveness. However, such approaches are grammar-blindly and they are not effective in fuzzing JavaScript engines.

BuzzFuzz [29], TaintScope [55], and VUzzer [48] try to identify the relationship between the input and the target program, which is further used to improve the mutation. Such approaches reduce the mutation space and avoid the blindness of conventional fuzzing.

Generation-Based Methods. For generation-based methods, the test cases are generated according to the rules that are manually written by humans. Peach [12] and Sulley [11] are the two of the most representative generation-based fuzzing frameworks. These two frameworks depend on the customized formatting specifications, *e.g.*, the logical relationships between the data packets need to be specified before testing the network service software.

What's more, some researchers proposed anti-fuzzing techniques[35] to prevent adversaries from looking for zero-day vulnerabilities. What's more, some scholars proposed the modularized fuzzing framework FOT [24], which can easily integrate new methods.

7.2 Structural Test Cases Generation

In addition, researchers have also proposed some methods to generate test cases with complex structures. UMLTEST [43] uses the UML to represent the behavior of an object and generates test cases. ADLscope [22] utilizes the formal specification specified in Sun Microsystems' Assertion Definition Language (ADL) of a program unit as the basis for test selection and test coverage measurement. But these methods do not consider the relationship between structures. To solve this problem, Korat [42] uses constraint-based method to generate structurally complex test inputs. Korat performs a systematic search of the predicate's input space and generates bounded-exhaustive testing for programs. In recent research, Pham et al. propose AFLSmart [47], which leverages a high-level structural representation of the seed files and applies higher-order mutations on the seeds to generate new test inputs. However, these methods rely on manual efforts to write various specifications. In contrast, SoFi aims to automatically generate semantically valid test inputs.

7.3 JavaScript Fuzzing

JavaScript has been widely used in many software. Many works (*e.g.*, [4, 5, 8, 9]) proposed to test JavaScript engines. jsfunfuzz [9] and domato [8] rely on pre-defined rules to generate test cases. Such methods could guarantee syntax correctness. However, they rely on some hard-coded or manually-specified rules, which may be incomplete and affect the usefulness of the fuzzer.

In order to generate structured inputs more efficiently, some early works take syntax and semantic into consideration [19, 41, 57–59]. LangFuzz [34] parses the JavaScript seed into an AST, and then generates test cases by replacing the AST fragments with the information obtained from a seed corpus. Superion [54] and NAUTILUS [18] combine the AST-based mutation with the coverage-guides approaches. IFuzzer [51] uses genetic algorithms to optimize the mutation of the AST. However, none of these approaches uses reflection to improve the diversity of test cases.

To further improve the semantic correctness of generated test cases, researchers recently focus on semantic-aware fuzzers [33, 45, 51, 53]. Skyfire [53] learns the probabilistic context-sensitive grammar from an existing seed corpus to be used as the semantics of the programs for fuzzing. Some other works [28, 32, 37, 49] also

apply machine learning to learn the semantic rules of the test cases from a large database, which can be used to generate more test cases. CodeAlchemist [33] generates semantically valid test cases based on type analysis. However, it could not effectively analyze the code fragments that are not executed. Moreover, the type analysis is path-insensitive which may incorrectly check the availability of variables. DIE [44] proposes an aspect-preserving mutation to preserve the desirable properties of the seeds. A structure and type preserving mutation strategy is proposed to improve the input validity.

Different from the above methods, SoFi can automatically obtain attributes and methods of variables based on the dynamic reflection, which avoids the necessity of manual rule definition and can generate some new statements that are unable to obtain based on AST mutation. At the same time, SoFi adopts a set of type analysis, including dynamic, static, and historical test case learning to ensure the accuracy of type inference. A path-sensitive analysis is introduced to ensure the consistency of variables as much as possible. Such methods could improve the semantic validity of the generated inputs. In addition, we have also implemented an automatic test case repair algorithm to improve the usability of invalid test cases.

8 CONCLUSION

In this paper, we proposed SoFi, a novel semantic-aware fuzzer for discovering vulnerabilities of JavaScript engines. SoFi adopts an accurate pre-processing analysis and automatic fix to improve the semantic validity of generated test cases. Moreover, the reflection is adopted to enhance the diversity of test cases by identifying unseen methods of objects. Our evaluation demonstrates that SoFi outperforms the state-of-the-art techniques in terms of bug finding and code coverage. In total, SoFi discovered found 51 bugs in 9 popular JavaScript engines and 28 of them have been confirmed or fixed by developers. In the future, we plan to 1) introduce more mutation such that more diverse test cases can be generated and 2) extend the existing repair rules such that the semantic validity of test cases is further improved.

ACKNOWLEDGMENTS

This research was supported in part by the National Natural Science Foundation of China (Grant No. 61802394, U1836209, 62032010), National Key Research and Development Program of China (2020YFB1005704), and Strategic Priority Research Program of the CAS (XDC02040100). And this research is also supported by the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001.

REFERENCES

[1] american fuzzy lop. https://lcamtuf.coredump.cx/afl/.
[2] benjamn/ast-types: Esprima-compatible implementation of the mozilla js parser api. https://github.com/benjamn/ast-types.
[3] cesanta/mjs: Embedded javascript engine for c/c++. https://github.com/cesanta/mjs.
[4] dharma. https://github.com/MozillaSecurity/dharma.
[5] esfuzz. https://github.com/estools/esfuzz.
[6] Esprima. https://esprima.org/.
[7] estools/escodegen: Ecmascript code generator. https://github.com/estools/escodegen.
[8] googleprojectzero/domato: Dom fuzzer. https://github.com/googleprojectzero/domato.
[9] Mozillasecurity/funfuzz: A collection of fuzzers in a harness for testing the spidermonkey javascript engine. https://github.com/MozillaSecurity/funfuzz.
[10] Mujs. https://mujs.com/.
[11] Openrce/sulley: A pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley.
[12] Peach fuzzer: Discover unknown vulnerabilities. https://www.peach.tech/.
[13] Sofi. https://sites.google.com/view/sofi4js, 2021.
[14] tc39/test262: Official ecmascript conformance test suite. https://github.com/tc39/test262.
[15] tunz/js-vuln-db: A collection of javascript engine cves with pocs. https://github.com/tunz/js-vuln-db.
[16] Xs7 @ tc-39. https://www.moddable.com/XS7-TC-39.php.
[17] Zero day initiative — deconstructing a winning webkit pwn2own entry. https://www.thezdi.com/blog/2017/8/24/deconstructing-a-winning-webkit-pwn2own-entry.
[18] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.
[19] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. *ACM SIGPLAN Notices*, 52(6):95–110, 2017.
[20] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.
[21] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.
[22] Juei Chang and Debra J Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Software Engineering—ESEC/FSE'99*, pages 285–302. Springer, 1999.
[23] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. Muzz: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2325–2342, 2020.
[24] Hongxu Chen, Yuekang Li, Bihuan Chen, Yinxing Xue, and Yang Liu. Fot: A versatile, configurable, extensible fuzzing framework. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 867–870, 2018.
[25] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108, 2018.
[26] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.
[27] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. Memfuzz: Using memory accesses to guide fuzzing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 48–58. IEEE, 2019.
[28] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 95–105, 2018.
[29] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484. IEEE, 2009.
[30] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 206–215, 2008.
[31] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20–27, 2012.
[32] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59. IEEE, 2017.
[33] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.
[34] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzcodealchemisting with code fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.
[35] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. Fuzzification: Anti-fuzzing techniques. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1913–1930, 2019.
[36] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.
[37] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language model-guided javascript engine fuzzer. *arXiv preprint arXiv:2001.04107*, 2020.

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

[39] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.

[40] 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.

[41] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. Parser-directed fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 548–560, 2019.

[42] Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. Korat: A tool for generating structurally complex test inputs. In *29th International Conference on Software Engineering (ICSE'07)*, pages 771–774. IEEE, 2007.

[43] Jeff Offutt and Aynur Abdurazik. Generating tests from uml specifications. In *International Conference on the Unified Modeling Language*, pages 416–429. Springer, 1999.

[44] 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.

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

[46] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[47] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[48] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[49] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. Mtfuzz: Fuzzing with a multi-task neural network. *arXiv preprint arXiv:2005.12392*, 2020.

[50] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[51] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*, pages 581–601. Springer, 2016.

[52] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 999–1010. IEEE, 2020.

[53] 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.

[54] 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.

[55] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.

[56] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 765–777, 2020.

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

[58] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

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