CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines

HyungSeok Han
KAIST
hyungseok.han@kaist.ac.kr

DongHyeon Oh
KAIST
donghyeon.oh@kaist.ac.kr

Sang Kil Cha
KAIST
sangkilc@kaist.ac.kr

***Abstract*—JavaScript engines are an attractive target for attackers due to their popularity and flexibility in building exploits. Current state-of-the-art fuzzers for finding JavaScript engine vulnerabilities focus mainly on generating syntactically correct test cases based on either a predefined context-free grammar or a trained probabilistic language model. Unfortunately, syntactically correct JavaScript sentences are often semantically invalid at runtime. Furthermore, statically analyzing the semantics of JavaScript code is challenging due to its dynamic nature: JavaScript code is generated at runtime, and JavaScript expressions are dynamically-typed. To address this challenge, we propose a novel test case generation algorithm that we call semantics-aware assembly, and implement it in a fuzz testing tool termed CodeAlchemist. Our tool can generate arbitrary JavaScript code snippets that are both semantically and syntactically correct, and it effectively yields test cases that can crash JavaScript engines. We found numerous vulnerabilities of the latest JavaScript engines with CodeAlchemist and reported them to the vendors.**

I. Introduction

JavaScript (JS) engines have become a core component of modern web browsers as they enable dynamic and interactive features on web pages. As of July of 2018, approximately 94.9% of all the websites use JS [34], and almost all smartphones are equipped with a JS-enabled browser today.

The growing popularity of JS also means that JS engines are an appealing target for attackers. As developers attempt to improve performance and functionality of modern JS engines, they continuously introduce new security vulnerabilities. Such vulnerabilities can be exploited by an attacker to compromise arbitrary victim machines [19], [20] that are potentially behind a firewall. In this case, the attacker simply needs to entice the victim to visit a malicious web page containing a malicious JS snippet.

As such, there has been much research on finding JS engine vulnerabilities. LangFuzz [17] is one of the most successful fuzzers in this field, finding more than 2,300 bugs in JS engines since 2011 [16]. LangFuzz initially parses sample JS files, referred to here as JS seeds, and splits them into code fragments. It then recombines the fragments to produce test cases, i.e., JS code snippets. Another successful JS engine fuzzer is jsfunfuzz [27], which does not require any sample files unlike LangFuzz. Instead, it randomly generates syntactically valid JS statements from a JS grammar manually written for fuzzing. Although this approach requires significant manual effort to implement language production rules, it is extremely efficient in finding JS engine vulnerabilities: it has found more than 2,800 bugs since 2006 [16].

Although successful, current state-of-the-art JS engine fuzzers suffer from generating semantically valid JS code snippets. According to our preliminary study, more than 99% of test cases produced by jsfunfuzz raise a runtime error after consuming only three JS statements. Given that LangFuzz does not consider the JS semantics while associating the code fragments, it has more or less the same problem. For example, test cases generated from current fuzzers may refer to variables that are not defined in the current execution context.

One may argue that it is possible to extend jsfunfuzz to handle full-fledged JS semantics, but writing a complete grammar for fuzzing requires effort nearly identical to that required when writing a JS engine, which is not feasible in practice. To mitigate requirement of the manual effort while reducing the number of invalid JS test cases to generate, jsfunfuzz makes JS errors silent by wrapping JS code snippets with `try-catch` statements. However, such an approach does not resolve the root cause of JS runtime errors, and it may change the semantics of JS code snippets. For instance, the Proof of Concept (PoC) exploit snippet of CVE-2017-11799 shown in Figure 1 triggers the vulnerability only without a `try-catch` statement as a `try-catch` block suppresses the JIT optimization, which is critical to trigger the vulnerability.

Skyfire [35] and TreeFuzz [24] partially learn JS language semantics by building probabilistic language models from a corpus of JS seed files, and they use the models to generate test cases. However, these approaches largely rely on the accuracy of the language models, and they currently suffer from handling the complex type system of JS language, meaning they are highly likely to produce semantically invalid test cases. For example, consider the JS statement `x.toUpperCase()`. When the variable `x` is not a string, and it does not have a method `toUpperCase`, we will observe a type error when executing the statement. Unraveling such language semantics with a language model is difficult in practice.

In this paper, we propose a novel test case generation technique that we call *semantics-aware assembly*, which can systematically generate JS code snippets that are both syntactically and semantically correct in a fully automatic fashion.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23263
www.ndss-symposium.org

The crux of our approach is to break JS seeds into fragments that we refer to as code bricks. Each code brick is tagged with a set of constraints representing in which condition the code brick can be combined with other code bricks. We call such a condition as an *assembly constraint*. Specifically, we compute which variables are *used* and *defined* in each code brick using a classic data-flow analysis [1], and dynamically figure out their types. We merge code bricks only when the used variables in each code brick are properly defined from the other code bricks in front, and their types match. Unlike LangFuzz where it joins arbitrary code fragments together as long as the language syntax allows it, the assembly constraints naturally help us follow the language semantics when interlocking code bricks. We note that some test cases generated from semantics-aware assembly may still throw a runtime error as it may over-approximate the assembly constraints. However, our empirical study shows that our technique can drastically reduce the likelihood of encountering runtime errors from the generated test cases compared to the state-of-the-art JS engine fuzzers.

Furthermore, semantics-aware assembly does not require any manual effort for implementing language grammars because it learns JS semantics from existing JS seeds. The similar intuition is used by LangFuzz, but our focus is not only on resolving syntactic errors, but also on semantic errors unlike any other existing JS engine fuzzers.

We implement semantics-aware assembly on a fuzzing tool that we refer to as CodeAlchemist, and evaluate it on four major JS engines: ChakraCore of Microsoft Edge, V8 of Google Chrome, JavaScriptCore of Apple Safari, and SpiderMonkey of Mozilla Firefox. Our tool was able to find 4.7× more unique crashes than jsfunfuzz in one of our experiments. It also discovered numerous previously-unknown security vulnerabilities from the latest JS engines at the time of writing.

Our main contributions are as follows.

1) We present semantics-aware assembly, a novel technique for fuzzing JS engines. The proposed technique can produce random yet semantics-preserving JS code snippets during a fuzzing campaign.
2) We implement our idea on a prototype called CodeAlchemist. To the best of our knowledge, CodeAlchemist is the first semantics-aware JS engine fuzzer.
3) We evaluate CodeAlchemist on four major JS engines, and found a total of 19 bugs including 11 security bugs. We reported all of them to the vendors.

II. Background

In this section we start by describing the characteristics of JS language and its type system. We then discuss the meaning of JS runtime errors that may occur while evaluating JS code. Since our design goal is to minimize such errors while generating JS test cases, it is essential to understand what are they, and what are their implications.

A. JavaScript and the Type System

JS is a dynamic programming language. Thus, JS programs can modify the type system at runtime, and can dynamically generate code on the fly, e.g., the infamous `eval` function evaluates JS code represented as a string. The dynamic nature of the language significantly hurts the precision of static analyses [29].

```
//try {
    class MyClass {
        constructor() {
            this.arr = [1, 2, 3];
        }
        f() {
            super.arr = [1];
            this.x;
        }
    }
    let c = new MyClass();
    for (let i = 0; i < 0x10000; i++) {
        c.f();
    }
//} catch (e) {}
```

Fig. 1: A PoC JS snippet that triggers CVE-2017-11799 in ChakraCore. If the `try-catch` statement, which is currently commented out, is used, the code does not crash ChakraCore.

There are seven kinds of primitive types in JS: `Undefined`, `Null`, `String`, `Boolean`, `Symbol`, `Number`, and `Object`. An `Object` type is simply a collection of properties. Since JS is a dynamically-typed language, variables can be assigned with any type at runtime. For instance, one can assign the number 42 to a string variable `s` with no problem:

```
var s = 'string variable';
s = 42; // This is valid
```

Unlike classic object-oriented languages such as C++ and Java, JS is a prototype-oriented language, which means that an object instance A can inherit properties of another object instance B at runtime by simply setting B as a *prototype* of A. Any object instances in JS have a `__proto__` property, which can be accessed and modified at runtime. Consider the following JS statement as an example:

```
var arr = new Array(42);
```

After evaluating the above statement, the initialized array variable `arr` will have the `Array` object as its prototype. By accessing the property (`arr.__proto__`), we can easily figure out which prototype is used by an object instance at runtime. Additionally, we can also dynamically assign a new prototype to an object instance during a program execution. Since the type system of JS can dynamically change, it is likely to encounter runtime errors during the evaluation of JS statements.

B. JavaScript Runtime Errors

Even a syntactically valid JS snippet can raise runtime errors, which is indeed the key motivation to our research. There are five kinds of native runtime errors defined in the ECMAScript standard [9] (corresponding object names are in parentheses): syntax error (`SyntaxError`), range error (`RangeError`), reference error (`ReferenceError`), type error (`TypeError`), and URI error (`URIError`). Figure 2 presents sample statements that can raise them. Each line in the figure is independent to each other, and can run separately to throw a specific runtime error.

```
1 eval('break'); // SyntaxError
2 var r = new Array(4294967296); // RangeError
3 u; // ReferenceError
4 var t = 10; t(); // TypeError
5 decodeURIComponent('%'); // URIError
```

Fig. 2: Sample statements for runtime errors. Each line throws a specific runtime error when it is individually evaluated.

Syntax errors trigger when a JS engine interprets syntactically invalid code. Since JS is a dynamic language, code that looks syntactically correct can still raise a syntax error at runtime. For example, Line 1 in Figure 2 presents a case where a `break` statement is evaluated on the fly with the `eval` function. Note that the statement itself is syntactically correct, but a syntax error is raised when it is evaluated because it is not within a loop in the current context.

Range errors happen when we try to use a value that is not in the allowed range. For example, Line 2 shows a syntactically valid JS statement, which will throw a range error because the `Array` constructor only accepts a number less than $2^{32} - 1$, and $4294967296 = 2^{32}$.

Reference errors occur when accessing an undefined variable. The variable `u` in Line 3 is used without any prior definition. According to our study, this is the most common error that we can detect while fuzzing JS engines.

Type errors arise when the actual type of a value is different from the expected type. For example, Line 4 declares an integer variable `t`, but then, we consider the variable as a function, and make a function call. As a result, we will encounter a type error after executing the line.

Finally, URI errors appear during the execution of global URI functions when they were used in a way that is incompatible with their definition. Line 5 raises a URI Error because the given parameter string is not a valid URI.

In addition to the native errors, there can be runtime errors defined by programmers, which we refer to as a *custom error* in this paper. One can define custom errors by instantiating the `Error` object, and can raise them with a `throw` statement. Our focus in this paper is on reducing the number of native runtime errors, but not custom errors as they are a part of the JS semantics anyways.

III. Motivation

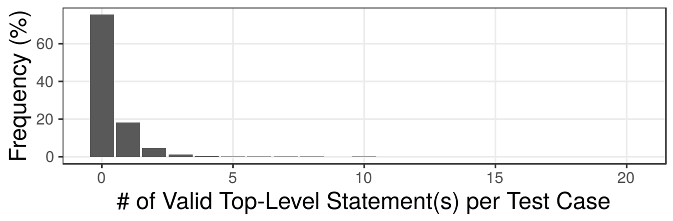
Our research is inspired by a preliminary study that we performed with jsfunfuzz [27], one of the state-of-the-art JS engine fuzzers. We chose jsfunfuzz because it is historically the most successful JS engine fuzzer that is open-sourced. LangFuzz [17], for instance, is not publicly accessible.

We ran jsfunfuzz on the four major JS engines, i.e., ChakraCore, V8, JavaScriptCore, SpiderMonkey, and observed interesting phenomena: (1) we encountered a runtime error for every JS file that jsfunfuzz generated; and (2) each of the file returned a runtime error after evaluating only few statements. Particularly, we were able to catch a runtime error in 99.5% of the cases with only three or less top-level [1] statements.

[1]Each top-level statement generated from jsfunfuzz included 2.5 statements on average.

Kind	**# of Occurrences**			
Syntax Error	18,200	17,429	17,998	17,135
Range Error	310	285	328	308
Reference Error	78,294	79,116	78,401	78,935
Type Error	3,196	3,169	3,273	3,507
URI Error	0	0	0	0
Custom Error	0	1	0	115
Total Count	100,000	100,000	100,000	100,000

(a) Classification of runtime errors encountered while fuzzing the four major JS engines with jsfunfuzz for 100,000 iterations. The four engines are ChakraCore, V8, JavaScriptCore, and SpiderMonkey.



(b) The frequency of JS files (out of 100,000 generated files) over the number of valid top-level statement(s).

Fig. 3: Our preliminary study on jsfunfuzz.

We counted how many runtime errors that we can catch while evaluating 100,000 dynamically generated JS code snippets on each of the engines. Note that jsfunfuzz generates a stream of a potentially infinite number of JS statements until it finds a crash while suppressing any runtime error by wrapping JS code blocks with `try-catch` statements as appeared in the comments in Figure 1. For the purpose of this study, we ran jsfunfuzz on each JS engine for 20 fuzzing iterations. This means that we used jsfunfuzz to generate a sequence of 20 JS code blocks, which have 2.5 statements on average, wrapped with a `try-catch` statement, and stored the entire sequence to a file. We collected 100,000 of such JS files for each engine, and then removed `try-catch` statements from the files, so that we can immediately detect a runtime error while evaluating them. As a consequence, *all* the generated JS files produced a runtime error when evaluated.

Figure 3a summarizes the result. We found on average 78.7% and 17.7% of the JS files raised a reference error and a syntax error, respectively. In theory, grammar-based fuzzers such as jsfunfuzz should not produce any syntax error unless they produce some dynamically changing code, e.g., `eval`. However, we observed that most of the JS files were throwing a syntax error without dynamically modifying code. For example, we observed that jsfunfuzz can produce JS statements with a mismatching bracket. This is because jsfunfuzz has a manually written grammar, which may contain incorrect or incomplete production rules. And this result highlights the difficulty of writing grammars for fuzzing, which motivates one of our design goal: our fuzzer should *automatically* generate JS test cases while minimizing runtime errors without manually written grammar.

We further analyzed the result to check how many valid top-level statements—we counted a block statement as one regardless of how many nested statements it has—jsfunfuzz was able to evaluate for each JS file until it hits a runtime error. As we discussed, all the JS files from jsfunfuzz threw a runtime error when evaluated, but the first few statements in each JS file were indeed valid. Figure 3b presents the histogram of the number of valid top-level statements evaluated for each JS file with jsfunfuzz. Notably, more than 75% of the JS files threw a runtime error at the very first top-level statement, i.e., there was no semantically valid JS statement. And 99.5% of the JS files triggered a runtime error after evaluating three or less JS top-level statements. That is, only a few top-level statements produced by jsfunfuzz were valid at runtime, which is indeed the key observation that motivated our research.

The aforementioned observations point out a primary challenge in JS engine fuzzing: automatically generating semantically valid JS code snippets during a fuzzing campaign. In this paper, we address the challenge by introducing a novel fuzzing technique, called semantics-aware assembly.

IV. Overview

In this section, we start by introducing semantics-aware assembly, a novel test case generation algorithm for JS engine fuzzing. We then outline the overall architecture of CodeAlchemist, which implements semantics-aware assembly. Finally, we describe how CodeAlchemist generates valid JS test cases by stepping through each step it performs on a running example.

A. Semantics-aware Assembly

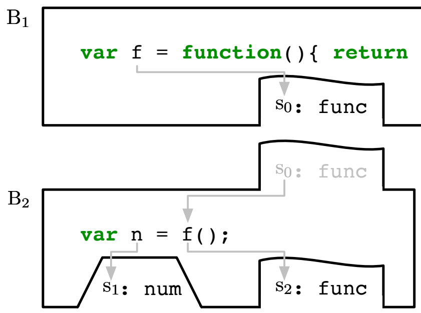
The primary challenge of CodeAlchemist is to generate test cases, i.e., JS code snippets that are both syntactically and semantically valid. To address the challenge, we propose *semantics-aware assembly*, a novel test case generation algorithm for JS engine fuzzing. The key intuition behind our approach is to fragmentize JS seeds into a set of combinable building blocks that we call *code bricks*. A code brick represents a valid JS Abstract Syntax Tree (AST). Therefore, a code brick itself can be evaluated by a JS engine. For example, a JS statement can become a code brick, and a block of statements (`BlockStatement`) can also become a code brick.

Each code brick can be annotated with an *assembly constraint*, which is a set of rules to follow when interconnecting the code brick with other code bricks. Specifically, an assembly constraint encodes two conditions: a precondition and a postcondition. A precondition is a set of variable symbols as well as their types that are required to be defined to execute the code brick without a runtime error.

A postcondition describes what kinds of variables are available, i.e., *defined*, at the end of the code brick after evaluating it. Given a code brick B, which does not have any preceding code brick in front, any code brick $B' \neq B$ can be a valid candidate that can be placed next to B if the postcondition of B satisfies the precondition of B'. Starting from a small code brick, semantics-aware assembly repeatedly extends it by merging it with additional code bricks. Note, by definition, a code brick can always become a valid JS code snippet.

```
var f = function (){ return 42; };
var n = f();
```

(a) A sample JS seed with two statements. We assume that we create one code brick for each statement, respectively.



(b) Two code bricks obtained from the seed. The teeth and holes on each code brick represent the assembly constraints.

Fig. 4: Code bricks with assembly constraints.

Suppose that CodeAlchemist splits the sample seed in Figure 4a into two initial code bricks that contain only the first and the second statement, respectively. Figure 4b shows the resulting code bricks. The first code brick B_1 has no precondition, but it has a postcondition with a function symbol s_0, which indicates any code brick calling a function can follow. The second code brick B_2 requires a definition of a function symbol s_0 in its precondition. It also has its postcondition that defines two symbols s_1 and s_2 as a number and a function, respectively.

We can use B_1 at the beginning, but not B_2 because of its assembly constraint: the precondition of B_2 requires a function symbol to be defined, but there is no preceding code brick in front of it. However, we can append both B_1 and B_2 to B_1, as the postcondition of B_1 can satisfy the precondition of B_1 and B_2. When two code bricks merge, unmatched symbols from the postcondition of the front brick propagate to the resulting code brick. For example, when we merge two B_1 code bricks, the final code brick will have a postcondition with two function symbols.

There are several design challenges to overcome in order to make semantics-aware assembly practical. We discuss the challenges in detail and show how we address them in §V. First, we need to decide how to break JS code into code bricks (§V-A). Second, we want to maintain a pool of code bricks while minimizing its size and preserving its semantics (§V-B). Third, we should rename symbols in code bricks when we assemble them so as to avoid potential reference errors (§V-C). Fourth, we need to figure out data dependencies between variables in each code brick in order to compute assembly constraints (§V-D). Fifth, the precision of assembly constraints largely depends on the quality of our type analysis. Thus, we should design an effective way to infer types of variables (§V-E). Lastly, we need to devise an effective way to combine code bricks with assembly constraints to build highly-structured code snippets (§V-F).

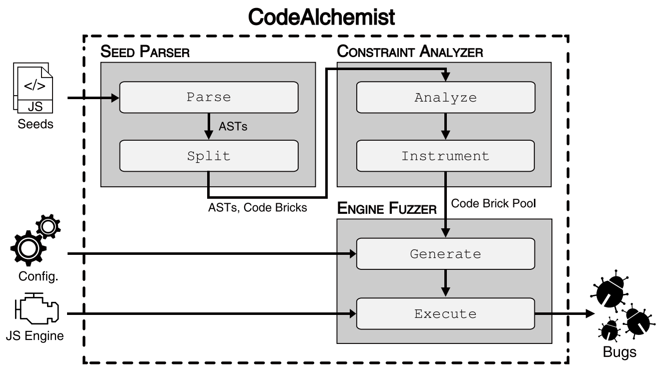


Fig. 5: CodeAlchemist Architecture.

B. *CodeAlchemist Architecture*

Figure 5 depicts the architecture of CodeAlchemist. At a high level, it takes in as input a JS engine under test, a set of JS seed files, and a set of user-configurable parameters, and it outputs a set of bugs found in the engine. CodeAlchemist consists of three major components: SEED PARSER, CONSTRAINT ANALYZER, and ENGINE FUZZER. The SEED PARSER module breaks given JS seeds into a set of code bricks. The CONSTRAINT ANALYZER module then infers assembly constraints for each code brick, and annotates them with the computed assembly constraints, which ultimately constitute a code brick pool. Finally, the ENGINE FUZZER module assembles the code bricks from the pool based on their assembly constraints to generate test cases and to execute the generated test cases against the target JS engine.

1) SEED PARSER*:* This module first parses each JS seed down to an AST based on the ECMAScript language specification [9]. The `Parse` function returns an AST from a given seed as long as it is syntactically correct. To filter out semantically unique code bricks, the `Split` function breaks the ASTs into code bricks and normalizes the symbols in them. All the broken code bricks should represent a valid AST, although they are not tagged with assembly constraints yet.

2) CONSTRAINT ANALYZER*:* This module figures out an assembly constraint for each of the fragmentized code bricks. First, the `Analyze` function recognizes which symbols are *used* and *defined* in each code brick using a classic data-flow analysis [1]. The `Instrument` function then traces types of the variables by dynamically instrumenting code bricks. As a result, CONSTRAINT ANALYZER returns a set of code bricks, each of which is tagged with an assembly constraint. We call such a set as a *code brick pool*, which is later used to generate test cases, i.e., JS code snippets, for fuzzing.

3) ENGINE FUZZER*:* Now that we have code bricks to play with, the ENGINE FUZZER module uses them to fuzz the target JS engine. Specifically, the `Generate` function iteratively assembles code bricks based on their assembly constraints in order to generate test cases. It also takes a set of user-configurable parameters which adjusts the way of combining the code bricks (see §V-F). Finally, the `Execute` function executes the target JS engine with the generated test cases. If the engine crashes, it stores the corresponding test case (a JS file) on a file system.

```
var n = 42; // Var1
var arr = new Array(0x100); // Var2
for (let i = 0; i < n; i++) // For3-0, For3-1
{ // Block4
  arr[i] = n; // Expr5
  arr[n] = i; // Expr6
}
```

(a) An example JS code snippet used as a seed.

```
var s0 = new Array(0x100); // Var2
var s1 = 42; // Var1
for (let s2 = 0; s2 < s1; s2++) { // For3-1
  for (let s3 = 0; s3 < s2; s3++) { // For3-0
    s0[s3] = s2;
    s0[s2] = s3;
  }
}
```

(b) A generated code snippet from the seed.

Fig. 6: A running example.

C. *Running Example*

We now discuss the detailed procedure of CodeAlchemist step by step. Suppose CodeAlchemist takes in the code snippet shown in Figure 6a as a seed. At a high level, CodeAlchemist will repeatedly produce test cases based on the semantic structure that it learned from the seed. Figure 6b presents one of such test cases.

First, CodeAlchemist parses the given seed to obtain an AST. It then breaks the AST into a set of code bricks. In the current implementation of CodeAlchemist, we fragmentize an AST in the granularity of JS statements. Figure 6a presents in the comments what kind of code bricks are generated for each statement. Specifically, the seed is broken into seven distinct code bricks: two code bricks for the variable declaration statements (`Var1, Var2`); a code brick for the whole `for`-loop statement and another with an empty body (`For3-0, For3-1`); a code brick for the body of the loop itself (`Block4`); and two code bricks for the expression statements (`Expr5, Expr6`). Note that the body of `For3-1` is empty and it can be used to generate diverse `for`-loops, whereas `For3-0` represents the whole `for`-loop statement. For example, we can construct nested `for`-loops with `For3-1`, but not with `For3-0`.

Next, CodeAlchemist normalizes all the identifiers in the code bricks, and deduplicates them to minimize the number of code bricks to consider when assembling them. We exclude built-in symbols such as `Array` from normalization to preserve the semantics of code bricks. In our case, `Expr5` and `Expr6` are the same code brick as they appear in the form of `s0[s1] = s2`, where `s0, s1`, and `s2` are placeholders for three distinct variables. Thus, we will have a total of six code bricks in our pool after this step.

Now that we have obtained a set of unique code bricks, CodeAlchemist annotates each of them with an assembly constraint. To compute assembly constraints, we first figure out which variables are used and defined in each code brick with a static data-flow analysis. Note that we will not use normalized symbols in this example to ease the explanation.

As an example, let us consider `Var1`, which does not have any used variable, but has one defined variable `n`. If we denote the use-def variables by $U \rightarrow$ `Code Brick` $\rightarrow D$, where U is a set of used variables and D is a set of defined variables, then we can obtain the use-def variables for each code brick as follows.

1) {} → `Var1` → {`n`}
2) {} → `Var2` → {`arr`}
3) {`n,arr`} → `For3-0` → {`n,arr`}
4) {`n`} → `For3-1` → {`n,i`}
5) {`i,n,arr`} → `Block4` → {`i,n,arr`}
6) {`i,n,arr`} → `Expr5` → {`i,n,arr`}

We note that `For3-0` corresponds to the whole `for`-loop including its body, and it does not define any new variable, because the variable `i` is not *live* after escaping from the `for`-loop. However, the same variable `i` is considered as a defined variable for `For3-1` because it is accessible within the loop body, i.e., it is *live* at the entry of the loop body. The `For3-1` code brick is used to create a new `for`-loop from scratch by filling in its body with other code bricks.

CodeAlchemist then rewrites the given seed file to log types of all the variables in each of the code bricks. Since we split off a seed in the granularity of JS statements, we instrument every JS statement. By executing the rewritten seed, we can dynamically identify types of all the variables used or defined in each code brick. For example, we can figure out that the variable `i` and `n` are a `Number`, and the variable `arr` is an `Array`. CodeAlchemist annotates each code brick with the inferred type information, which forms an assembly constraint. For instance, the assembly constraint for `Expr5` is as follows.

$$\begin{array}{c}\text{Pre: }\{\texttt{s0: Number, s1: Number, s2: Array}\}\\ \downarrow\\ \texttt{Expr5}\\ \downarrow\\ \text{Post: }\{\texttt{s0: Number, s1: Number, s2: Array}\}.\end{array}$$

The assembly constraint indicates that one or more code bricks should precede `Expr5` because its precondition requires the three defined variables. After this step, our pool contains six annotated code bricks.

Finally, CodeAlchemist generates test cases by interlocking code bricks in the pool. It starts with an empty code brick B_0 that has an empty assembly constraint. Since B_0 has no postcondition, we can only append a code brick to B_0 if it does not have any precondition. There are two such code bricks: `Var1` and `Var2`. CodeAlchemist selects one at random, and in this example, CodeAlchemist picks `Var2`. It then creates a new code brick B_1 by merging B_0 with `Var2`, which simply results in `Var2` (Line 1 in Figure 6b).

The combined code brick B_1 has a postcondition that defines an `Array` variable. This means a code brick that uses an `Array` variable in its precondition can follow B_1. However, in our case, there are still two available candidates: `Var1` or `Var2`. For instance, `For3-0` is not available, because it depends on a `Number` variable as well as an `Array` variable. Suppose CodeAlchemist selects `Var1`, and appends it to B_1 to obtain a new code brick B_2 that contains two consecutive JS statements `Var2` and `Var1`. When we compute the postcondition of B_2, it contains two variables: an `Array` and a `Number` variable.

Therefore, there are four distinct code bricks that we can append to B_2: `Var1, Var2, For3-0`, and `For3-1`. In this example, CodeAlchemist randomly picks `For3-1` among the candidates, and connects it with B_2, which results in a new code brick B_3 containing two assignment statements followed by an empty for loop. We compute the postcondition of B_3, which has two `Number` variables (`s1` and `s2`) and an `Array` variable (`s0`). Therefore, at this point, any code bricks in our pool can be appended to B_3. Suppose CodeAlchemist randomly picks `For3-1` as the next code brick, and appends it to B_3 to generate a new code brick B_4, which represents the entire code snippet shown in Figure 6b. Assuming that we have set the maximum number of attempts for code brick generation to three, CodeAlchemist stops to expand the code brick at this point. CodeAlchemist repeats this process to fuzz the target engine until it hits a timeout.

V. CodeAlchemist Design

In this section, we present the design and implementation of CodeAlchemist in detail. CodeAlchemist enables semantics-aware test case generation for JS engines, which allows us to discover security vulnerabilities from the target engines.

A. Seed Fragmentization

In this paper, we have defined a code brick as a valid AST (see §IV-A). According to the ECMAScript standard [9], a JS AST is expressed as a set of statements and expressions, where a statement can recursively include other statements and expressions. For example, a `while` statement consists of a guard and a body statement, where the body statement can contain a group of other statements and expressions.

One straightforward way of fragmentizing ASTs is to follow the approach of LangFuzz [17], where it breaks an AST into a set of subtrees by making each of its non-terminal nodes in the AST as a root node of a subtree. However, as the authors admitted there can be many overlapped fragments in the pool if we consider every non-terminal node in the AST.

Another way is to split them in the granularity of JS expressions. Since a JS expression itself forms a valid AST, it can be a valid code brick too. Expression-level fragmentization results in smaller number of code bricks compared to the LangFuzz's approach, but it does not capture the high-level structure of JS code. For instance, it is not straightforward to generate a JS statement that contains a `for`-loop with expression-level fragmentization.

In our current implementation of CodeAlchemist, we fragmentize seeds in the granularity of JS statements. Since every code brick represents a valid statement, it is straightforward to interconnect them to make a valid JS code snippet: we can simply arrange them in a sequential order. Furthermore, statement-level fragmentization significantly reduces the number of code bricks to consider compared to that of the two aforementioned fragmentization approaches. However, it is not difficult to express complex semantic structure of JS with our code bricks.

The SEED PARSER module *recursively* traverses each statement in an AST, and returns a set of fragmentized code bricks. Since a JS statement can be nested, we need to recursively find every statement. While traversing an AST, we may encounter a block statement, which is a group of statements. For example, a body of a `while`-loop is a block statement. For every block statement, we make two separate code bricks: one with the whole original body, and another with an empty body. If the block statement has a guard, we make an additional code brick that contains the statement as a whole, but without the guard. For instance, consider the following `while` statement:

```
while (x) { i += 1; }
```

This statement will produce four code bricks in total. One code brick contains the whole `while`-loop including the body:

```
while (s0) { s1 += 1; } // Code Brick w/ body
```

Another code brick has the `while`-loop with an empty body:

```
while (s0) {} // Code Brick w/o body
```

Another code brick has the loop body without the guard:

```
{ s0 += 1; } // Code Brick w/o guard
```

The final code brick is found by the recursive traversal:

```
s0 += 1; // Last Code Brick
```

This is to increase the likelihood of generating test cases with a complex and nested control structure, while preserving the original semantic structure of given seeds. From our experiments, we observed that highly-constructed JS code tends to trigger security-related bugs of JS engines.

B. Code Brick Pool

CodeAlchemist builds a *code brick pool* that contains a unique set of code bricks obtained from various seed files. Since there can be syntactically different, but semantically the same code bricks, CodeAlchemist deduplicates them and only stores unique code bricks to the pool. Particularly, CodeAlchemist considers two code bricks to be the same, when they have the same AST except for their symbols. For instance, the two statements in Line 5 and Line 6 in Figure 6a are the same if we normalize their symbols.

CodeAlchemist also filters out several *uninteresting* code bricks from the pool. First, there are several built-in functions that hinder the fuzzing process when building a code brick pool, such as `crash` function on SpiderMonkey. The `crash` function will literally raise a `SIGSEGV` signal when executed. Therefore, if we have such a code brick that contains a call to `crash`, we may end up having too many false alarms from our fuzzer. We also exclude the `eval` function when building code bricks. Since CodeAlchemist currently cannot infer assembly constraints for dynamically generated code, we may encounter runtime errors when we combine code bricks that invoke the `eval` function. Finally, we also eliminate code bricks that can be considered as no-op, e.g., an expression statement with a literal (`42;`) is effectively a no-op.

To ensure that generated code bricks are syntactically correct, we evaluate all of them on the target JS engine once. If there is a syntax error, we remove such code bricks from our pool. From our empirical study, only 1.3% of code bricks generated from semantics-aware assembly result in a syntax error, and most of them are due to dynamically generated code using the `Function` object.

C. Semantics-Preserving Variable Renaming

CodeAlchemist renames variables of code bricks in two cases: (1) when it builds a code brick pool (recall §V-B), it deduplicates semantically the same code bricks by normalizing symbols in each code brick; and (2) when it assembles two code bricks, it renames symbols so that all the used variables can refer to variables of the same type.

However, there are pre-defined symbols that we cannot simply rename as we may break their semantics. In particular, each JS engine has pre-defined symbols, which are so-called built-in objects. For example, `RegExp` is a pre-defined object for a regular expression, and thus, we cannot simply change the symbol as it means a specific built-in JS object.

To rename symbols in code bricks while preserving their semantics, CodeAlchemist initially gets a list of built-in objects by executing the target JS engine once at startup. When renaming symbols in a code brick, we exclude symbols in the list. More specifically, we incrementally assign a sequential number for each unique symbol in the order of their appearance in the AST, and assign a new name for each symbol by prefixing "s" to its number. That is, we give a symbol name `s0`, `s1`, $\cdots$, `sn` to n distinct symbols in the code brick in the order of their appearance in the AST.

D. Data-Flow Analysis

The `Analyze` function statically performs a data-flow analysis to identify variables that are used and defined in each code brick. For a given code brick B, *defined* variables of B are the ones that are live at the exit point of B. We can compute such variables for a given code brick with a traditional live variable analysis [1] based on the control-flow graph of the code brick. We say variables in a code brick B are *used* variables if they do not have a reaching definition [1]. We compute used variables of a code brick by maintaining a use-def chain.

Since our analysis is path-insensitive as in traditional data-flow analyses, we may incorrectly judge defined and used variables from a code brick. For example, suppose there is a code brick that contains a single `if` statement where the true branch assigns a number to a variable `x`, and the false branch does not touch the variable `x` at all. Depending on the condition of the `if` statement, the variable `x` can be considered as defined or not. However, our analysis will always say the variable is defined. This means our analysis may incorrectly judge the assembly constraint of a given code brick. However, making the computation of assembly constraints precise with a path-sensitive analysis is beyond the scope of this paper.

E. Type Analysis

Recall from §II-A, the JS type system has only seven primitive types. If we only consider such primitive types, CONSTRAINT ANALYZER may over-approximate types of variables in each code brick as an `Object` type because nearly all objects in JS are instances of `Object` [21]. For example, any useful data structure types in JS such as `Array` and `Map` are inherited from the `Object` type.

The key problem here is that by over-approximating assembly constraints for code bricks, we may easily hit a runtime error when executing interconnected code bricks. For instance, suppose there are three code bricks each of which consists of a single JS statement as follows:

```
o = {}; // Code Brick A
a = new Array(42); // Code Brick B
s = a.sort(); // Code Brick C
```

If our type system can only distinguish primitive JS types, the code brick A will have a postcondition $\{s_0 : \texttt{Object}\}$, and the code brick C will have a precondition $\{s_0 : \texttt{Object}\}$. Since the precondition of C and the post condition of A match, CodeAlchemist will happily combine them into a code brick, which will produce the following JS code snippet:

```
s0 = {};
s1 = s0.sort();
```

Note, however, this will raise a runtime error (`TypeError`) when evaluated because the `sort` member function is not defined in `s0`.

To handle such problems, CodeAlchemist build our own type system that includes the seven primitive types as well as all the subtypes of `Object` defined in built-in constructors of the target JS engine. With our type system, we can distinguish between `Array` and `Object`. Thus, CodeAlchemist will not attempt to combine A and C from the above example.

CodeAlchemist dynamically infers types of the variables in each code brick by instrumenting the corresponding JS seed file. Particularly, the `Instrument` function rewrites a given JS seed by inserting type logging functions both at the beginning and at the end of each code brick. We recursively insert logging functions for every possible code brick of a given JS seed file. Each logging function takes in a list of variables as input, and returns a mapping from the variable names to dynamically inferred types. By executing the target JS engine with the rewritten seed, CodeAlchemist can compute assembly constraints for every code brick in the seed.

Note that two or more distinct seed files may have semantically the same code bricks after variable renaming. Although two code bricks from two different seed files have exactly the same AST, types of the variable in the code bricks may differ depending on their execution context. Let us consider the following code brick containing an `if` statement.

```
if (x < 42) y = 42;
else y = [];
```

From the above snippet, the variable `y` can be either a `Number` or an `Array` depending on the value of `x`. Suppose there are two seed files that contain the above code brick, and each seed executes the `if` statement with two distinct values 0 and 50 for `x`, respectively. Then the type of `y` can be either `Number` or `Array` depending on which seed we execute.

In this case, we give a union type for the variable `y` in the code brick. Specifically, when there are two or more seed files for a given code brick, we may have union types in

Algorithm 1: Code Generation Algorithm.

```
Input : A pool of code bricks ($P$),
        A code brick ($B$),
        The max number of iterations for code generation ($i_{\max}$),
        The probability of reinventing block statements ($p_{\text{blk}}$),
        The max number of statements in a block body ($i_{\text{blk}}$),
        The max nesting level for a block statement ($d_{\max}$).
Output: A generated code brick representing a test case.
1  function Generate($P$, $B$, $i_{\max}$, $p_{\text{blk}}$, $i_{\text{blk}}$, $d_{\max}$)
2      for $i = 1$ to $i_{\max}$ do
3          if RandProb() $< p_{\text{blk}}$ and $d_{\max} > 0$ then
4              $B' \leftarrow$ GenBlkBrick($P$, $B$, $p_{\text{blk}}$, $i_{\text{blk}}$, $d_{\max}$-1)
5          else
6              $B' \leftarrow$ PickBrick($P$, $B$)
7          $B \leftarrow$ MergeBricks($B$, $B'$)
8      return $B$
9  function GenBlkBrick($P$, $B$, $p_{\text{blk}}$, $i_{\text{blk}}$, $d_{\max}$)
10     $B' \leftarrow$ PickEmptyBlock($P$, $B$)
11     $B_0 \leftarrow$ GetDummyBrick($B$, $B'$)
12     $i \leftarrow$ RandInt($i_{\text{blk}}$)
13     $B'' \leftarrow$ Generate($P$, $B_0$, $i$, $p_{\text{blk}}$, $i_{\text{blk}}$, $d_{\max}$)
14     return MergeBricks($B'$, $B''$)
```

the assembly constraint. This is another instance where our technique lacks precision due to its path-insensitivity. As we discussed in §V-D, employing a path-sensitive way to compute assembly constraints is beyond the scope of this paper.

F. Code Brick Assembly

The `Generate` function of ENGINE FUZZER assembles code bricks in the code brick pool (P) to produce test cases. Algorithm 1 presents the pseudo code of it. There are four user-configurable parameters that `Generate` takes in.

$i_{\max}$	The maximum number of iterations of the generation algorithm. This parameter essentially decides how many top-level statements to produce.
p_{blk}	The probability of reinventing block statements. This parameter decides how often we generate block statements from scratch.
i_{blk}	The maximum number of iterations for generating a block statement. This parameter decides how many statements are placed within a newly generated block statement.
$d_{\max}$	The maximum nesting level for a reassembling block statement. When CodeAlchemist generates a new block statement from scratch, it limits the nesting depth of the block statement with $d_{\max}$, because otherwise, the generation algorithm may not terminate.

In this paper, we only focus on the first two parameters among the four ($i_{\max}$ and p_{blk}) while using the default values for the other two ($i_{\text{blk}} = 3$ and $d_{\max} = 3$). That is, we used the default values for the two parameters for all the experiments we performed in this paper. The default values were empirically chosen though several preliminary experiments we performed with varying i_{blk} and $d_{\max}$. For example, if i_{blk} and $d_{\max}$ were too large, generated test cases ended up having too complicated loops, which are likely to make JS engines stuck or hang when evaluated.

The algorithm starts with a code brick pool P, an empty code brick B, and the four parameters given by an analyst. In the `for`-loop of the algorithm (Line 2–7), CodeAlchemist repeatedly appends a code brick to B, and returns the updated B at the end, which becomes a new test case for fuzzing the target engine. The loop iterates for i_{max} times. Note our algorithm only appends code bricks: we currently do not consider code hoisting while stitching code bricks.

For every iteration, CodeAlchemist picks the next code brick B' to append, which can be either a regular JS statement or a reinvented block statement as we described in §V-A. It selects a reinvented block statement with the probability of p_{blk}, or a regular statement with the probability of $1 - p_{\text{blk}}$. The `RandProb` function in Line 3 returns a floating number from 0 to 1 at random.

The `GenBlkBrick` function in Line 4 randomly selects a code brick among the ones that have an empty block statement and the precondition of it satisfy the postcondition of B. It then fills up the body of the block statement, and returns the newly constructed code brick. On the other hand, the `PickBrick` function in Line 6 randomly picks a code brick, the precondition of which satisfies the postcondition of B, from the pool P. When there are multiple candidate code bricks, `PickBrick` randomly selects one with probability proportional to the number of unique symbols in the precondition. This is to increase the dependency among statements in the generated test cases as semantically complex code tends to trigger JS engine bugs more frequently.

Both Line 4 and Line 6 return a new code brick B' to use, which will be appended to B with the `MergeBricks` function in Line 7. To avoid reference errors, it replaces symbols in the precondition of B' with the symbols in the postcondition of B based on their types. When there are multiple symbols with the same type, we randomly select one of them, and replace its symbol. After renaming, CodeAlchemist recomputes the assembly constraint of the merged code brick in order to preserve the semantics of it.

The `GenBlkBrick` function builds a code brick for a block statement from scratch. The `PickEmptyBlock` function in Line 10 the pool P and the current code brick B maintained by CodeAlchemist as input, and returns a code brick B' in P that satisfies the following two conditions: (1) B' should contain an empty block statement, which may or may not include a guard, and (2) the precondition of B' should meet the postcondition of B. The `GetDummyBrick` function in Line 11 then extracts a random subset of the postconditions of B and B' in order to build a new postcondition c, and then create a dummy code brick B_0, where the postcondition of it is c. Next, CodeAlchemist generates a code brick B'' for the body of the block statement using the `Generate` function with the dummy code brick B_0.

Note that the `Generate` and `GenBlkBrick` are mutually recursive, and they allow us to generate nested block statements. We limit the nesting depth of a newly generated block by d_{max}. The `RandInt` function in Line 12 decides the maximum number of iterations to be used in generating the block body. It returns a random integer from 1 to i_{blk}. Finally, `GenBlkBrick` merges B' and B'', and returns the merged one as a new code brick containing a new block statement, which is potentially nested up to the depth d_{max}.

G. Implementation

We have implemented CodeAlchemist with 0.6K lines of JS code, 0.1K lines of C code, and 5K lines of F# code. We use JS for parsing and instrumentation, C for executing JS engines, and F# for the entire system. Our system heavily relies on the asynchronous programming feature of F#. Since our algorithm is written in a functional manner, i.e., no side-effects, it is straightforward to achieve concurrency.

To parse JS seeds, we use Esprima [15], which is a Node.js [22] library for parsing JS code. The parsed ASTs are then passed to the `Split` function, which is written in F#, as a JSON file. The counter part of JS parsing is to convert an AST to a JS code snippet. There exists a famous library in Node.JS, called escodegen [30], for that purpose. However, we implemented our own code generator in F# in order to reduce the communication cost between Node.JS and F#.

We also implemented our own JS library that includes several helper functions for figuring out types of variables in each code brick with dynamic instrumentation. Our system currently supports the seven primitive types and built-in types we mentioned in §V-E. The `Instrument` function (written in F#) rewrites given JS seeds in such a way that each code brick will call the instrumentation functions defined in the library.

Finally, the ENGINE FUZZER module is written in both F# and C. The `Generate` function, which is purely written in F#, generates test cases for fuzzing as we discussed in §V-F. The core of the `Execute` function, however, is written in C, in order to efficiently interact with native system functions. We make our source code public on GitHub: https://github.com/SoftSec-KAIST/CodeAlchemist.

VI. EVALUATION

We now evaluate CodeAlchemist to answer the followings:

1) Can CodeAlchemist generate semantically valid test cases? (§VI-B)
2) Does the fuzzing parameters such as i_{max} and p_{blk} affect the effectiveness of CodeAlchemist? If so, which values should we use? (§VI-C)
3) How does CodeAlchemist perform compared to the state-of-the-art fuzzers in terms of finding bugs? (§VI-D)
4) Can CodeAlchemist find real-world vulnerabilities in the latest JS engines? (§VI-E)
5) What do the vulnerabilities found by CodeAlchemist look? (§VI-F)

A. Experimental Setup

We ran experiments on a machine equipped with two Intel E5-2699 v4 (2.2 GHz) CPUs (88 cores) and 512 GB of main memory, which is operated with 64-bit Ubuntu 18.04 LTS. We selected the four major JS engines of the latest stable version as of July 10th, 2018: (1) ChakraCore 1.10.1; (2) V8 6.7.288.46; (3) JavaScriptCore 2.20.3; (4) SpiderMonkey 61.0.1. Note that we used only ChakraCore for the first three experiments due to our resource limit. We chose ChakraCore because it has

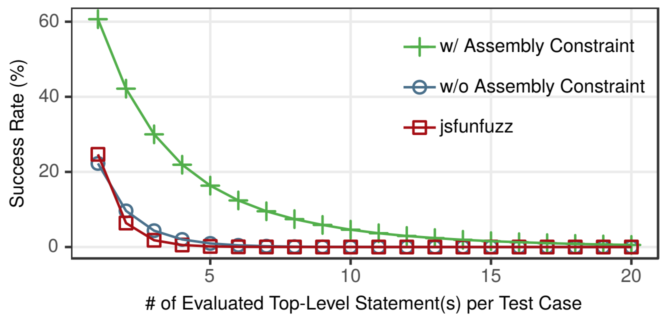


Fig. 7: The success rate (out of 100,000 test cases) over the number of evaluated top-level statement(s).

well-organized commit logs that specify which commit patches which CVE vulnerability. On the other hand, we used all the four JS engines for the rest of the experiments.

Seed Collection. Recall that semantics-aware assembly learns language semantics from JS seeds, and thus, having a reasonable set of seed corpus is essential. To gather JS seeds, we first downloaded (1) regression tests from repositories of the four major JS engines, and (2) test code snippets from Test262 [31], which is an official ECMAScript conformance test suite. From all the collected JS seed files, we filtered out some of them that are syntactically invalid or containing statements using engine-specific syntax. Finally, we were able to collect a total of 63,523 unique JS files from both test suites. In addition, we collected 169 PoC exploits, which can trigger previously known security vulnerabilities, by manually crawling bug trackers and GitHub repositories. In total, we gathered 63,692 unique JS seeds for our experiments.

Code Brick Pool. We first ran the SEED PARSER module to obtain 264,629 unique code bricks from the 63,692 seeds. We then ran the CONSTRAINT ANALYZER module to assign assembly constraints to the code bricks. Finally, we filtered out code bricks that contain uninteresting statements as we discussed in §V-B. Consequently, we gathered 49,800 code bricks in our code brick pool. Note that the reason why we have significantly less code bricks after the filtering step is because most regression tests use the `eval` function to compare evaluation results between two JS statements.

B. Validity of Generated Test Cases

Can semantics-aware assembly produce semantically valid test cases? To answer the question, we measured the number of runtime errors encountered by executing ChakraCore with test cases generated from CodeAlchemist. In this subsection, we say a test case is valid up to N statements if it does not raise any runtime error when the first N statements in the test case are evaluated. The *success rate* for N top-level statements is the rate between the number of valid test cases up to N top-level statements and the total number of evaluated test cases.

1) Comparison against jsfunfuzz: Recall from §III that all of the test cases obtained from jsfunfuzz threw a runtime error after evaluating only a few top-level statements. To compare CodeAlchemist against jsfunfuzz under the same condition, we configured CodeAlchemist to produce 20 top-level statements per each test case, i.e., $i_{\text{max}} = 20$. We then ran CodeAlchemist while randomly varying the block reinvention rate p_{blk} from 0.0 to 1.0 in order to obtain a set of 100,000 test cases (T_{ours}). We compared T_{ours} with the set of 100,000 test cases (T_{js}) that we used in §III, which are generated by jsfunfuzz. Particularly, we measured the success rate for N top-level statements by running ChakraCore with T_{ours} and T_{js}.

Figure 7 presents the success rates for 20 distinct N. The green line indicates the success rate of CodeAlchemist, and the red line indicates that of jsfunfuzz. When we evaluated only the first top-level statement for each test case ($N = 1$), 24.7% of test cases in T_{js} were valid, whereas 60.7% of test cases in T_{ours} were valid. That is, CodeAlchemist produced about 2.5× more valid test cases in this case. Similarly, when we evaluated the first three top-level statements for each test case ($N = 3$), 1.8% of test cases in T_{js} and 30.0% of test cases in T_{ours} were valid.

Overall, CodeAlchemist generated 6.8× more semantically valid test cases on average. Thus, we conclude that CodeAlchemist can produce substantially more valid test cases than jsfunfuzz, the current state-of-the-art JS engine fuzzer.

2) Effectiveness of Assembly Constraints: The crux of our system is that we can generate semantically valid test cases by assembling code bricks based on their assembly constraints. To justify our intuition, we ran a modified CodeAlchemist that does not produce assembly constraints for code bricks. That is, it produces code bricks and assembles them to generate test cases, but none of the code bricks will have its assembly constraint. Therefore, any code bricks can be interconnected to each other in this case. The lines in Figure 7 show that the success rate goes down as N increases because more statements are likely to have more type errors and reference errors. The modified CodeAlchemist (the blue line) has the same success rate as jsfunfuzz (the red line). However, it produced 5.7× less valid test cases on average compared to the unmodified CodeAlchemist: the unmodified CodeAlchemist has much less type errors and reference errors than the others. This result highlights the importance of assembly constraints, and confirms our intuition.

3) Note on the Success Rate: Although the success rate shows the overall quality of the generated test cases, there is a caveat: one can easily obtain a high success rate by producing meaningless test cases that are always semantically correct. Imagine a hypothetical fuzzer that always generates a sequence of variable assignments such as "`a = 1;`". This will never produce test cases that are semantically invalid, and thus, the success rate of this fuzzer is going to be always 100%. Nevertheless, we argue that our results in this subsection is still meaningful, because CodeAlchemist can generate test cases that trigger more number of vulnerabilities than the state-of-the-art fuzzers as we show in the rest of this paper.

In this regard, if we use a smaller range of p_{blk} from 0.0 to 0.5 for the same experiment we did in §VI-B1, CodeAlchemist produces twice more valid test cases than the result we showed. For our own purpose, CodeAlchemist needs to produce both semantically valid and highly-structured JS code snippets so that they can trigger various vulnerabilities in the target JS engine. Therefore, we need to find a good value for p_{blk} that can hit the sweet spot (see §VI-C2).

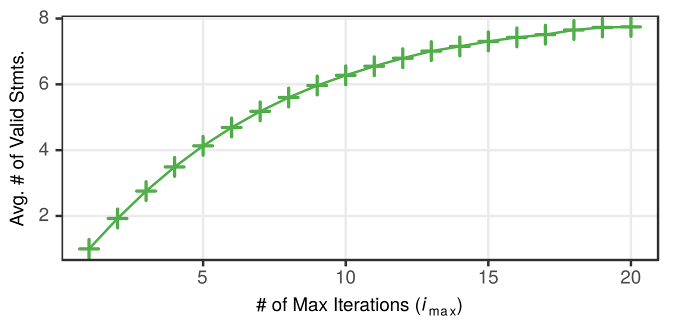


Fig. 8: The average number of valid top-level statements over $i_{\max}$. For each $i_{\max}$, we generated 100,000 test cases.

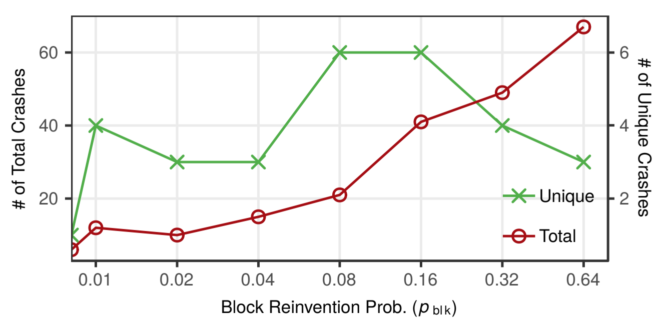


Fig. 9: # of bugs found over the probability of reinventing block statements (p_{blk}).

C. Choosing Parameters

Recall from §V-F that CodeAlchemist has four user-configurable parameters, and our focus in this paper is on $i_{\max}$ and p_{blk}. To understand the impact of using different parameter values, we conducted two independent experiments: (1) we measured how $i_{\max}$ affects the validity of generated test cases; and (2) we evaluated how p_{blk} changes the bug finding ability of CodeAlchemist.

1) Dose i_{max} Affect The Validity of Test Cases?: Intuitively, the number of generated statements in a test case can affect the validity of test cases because it is likely to have more complex semantic structure as we have more statements, and our analysis can be imprecise. To confirm this intuition, we ran CodeAlchemist with varying $i_{\max}$ values (from 1 to 20) to generate 100,000 test cases per each $i_{\max}$ value. Thus, we generated a total of 2,000,000 test cases for this experiment. We fix the block reinvention rate p_{blk} to zero for this particular experiment, because we wanted to see how $i_{\max}$ solely affect the validity of generated test cases. However, there exists the similar tendency even for larger p_{blk} because the likelihood of generating invalid test cases increases as we generate more statements in a test case.

Figure 8 illustrates the average number of valid statements in each set of test cases we generated with different $i_{\max}$. Note from the figure that the average number of valid statements of the test cases converges to 8 as $i_{\max}$ increases. Therefore, it is reasonable to choose 8 as the value of $i_{\max}$ as we can minimize the size of generated test cases while keeping the chance of generating a sequence of eight potentially valid statements. Smaller test cases are better than bigger ones as we can generate them more quickly.

2) Dose p_{blk} Affect The Bug Finding Ability?: Recall from §V-F, the probability of reinventing block statements (p_{blk}) decides how often we create block statements from scratch during the assembly process. The key intuition here is that security vulnerabilities often arise from a highly-structured JS code. Therefore, as we alter the value of p_{blk}, the bug finding effectiveness of CodeAlchemist may vary. To confirm this intuition, we ran CodeAlchemist with eight distinct p_{blk} from 0.0 to 0.64 (from 0% to 2^6%) on ChakraCore 1.10.1, and counted the number of crashes found in Figure 9. The red and the green line indicate the number of total crashes found and the number of unique crashes found, respectively.

First, we recognize that creating block statements from scratch significantly helps in finding more crashes. When $p_{\text{blk}} = 0.0$, we found the least number of unique crashes, but when p_{blk} is greater than 0, we found at least 3 unique crashes in each case. We believe this is because guarded block statements such as `for`-loops can initiate JIT optimization, which may trigger memory corruption in the end. Note that many recent CVEs were indeed assigned to JIT optimization vulnerabilities.

Second, increasing the block reinvention probability does not always help find unique bugs. For instance, when $p_{\text{blk}} = 0.16$, CodeAlchemist found 6 unique crashes and a total of 41 crashes. However, we found 3 unique crashes and a total of 67 crashes with $p_{\text{blk}} = 0.64$. This is mainly because (1) CodeAlchemist was stuck with infinite loops as the number of guarded block statements increases, and (2) several JIT optimization-based crashes potentially prevent us from observing other meaningful bugs.

From the above experiments it is obvious that finding an appropriate value for p_{blk} is important. Both 0.08 and 0.16 of p_{blk} found the most number of unique crashes, but we found more number of total crashes with 0.16. Thus, we decided to use $p_{\text{blk}} = 0.16$ for the rest of experiments.

D. Comparison against State-of-the-Art JS Engine Fuzzers

How does CodeAlchemist perform compared to existing JS fuzzers in terms of their bug finding ability? To answer this question, we compared CodeAlchemist against jsfunfuzz [27], which is the state-of-the-art fuzzers maintained by Mozilla, and IFuzzer [33], which is a variant of LangFuzz [17]. We first evaluated these three fuzzers to compare how many known CVEs and bugs they can find in an old version of ChakraCore. We then compared them on the latest stable version of the four major JS engines to see how many *unknown* bugs they can find.

1) Comparison on an Old ChakraCore: We selected an old version of ChakraCore (1.7.6), which was the first stable version of ChakraCore released after Jan. 1st, 2018. In order to run CodeAlchemist and IFuzzer, which require seed files to start with, we gathered JS seed files appeared before Jan. 1st, 2018 (as we did in §VI-A). Note we excluded regression tests released after Jan. 1st, 2018 because they may contain test cases that can trigger bugs in the old version of ChakraCore. For fair comparison, we also set the version of jsfunfuzz

TABLE I: The number of unique crashes found on ChakraCore 1.7.6 (released on Jan. 9th, 2018) by three different fuzzers.

	CodeAlchemist	**jsfunfuzz**	**IFuzzer**
# of Unique Crashes	7	3	0
# of Known CVEs	1	1	0

to be the latest one released before Jan. 1st, 2018. When we run CodeAlchemist, we use the following parameters: $i_{\text{max}} = 8, p_{\text{blk}} = 0.16$. We ran the three fuzzers for 2,112 CPU hours (24 hours with 88 cores) each, and compared the number of crashes found. Note that we counted only the crashes that involve memory corruption by manually verifying them, because JS engines sometimes intentionally raise a `SIGSEGV` signal, e.g., `WTFCrash` of JavaScriptCore [4].

Table I summarizes the number of unique crashes and known CVEs each fuzzer found. Note that CodeAlchemist found the most number of unique crashes, while IFuzzer was not able to find any crash. CodeAlchemist found twice more unique bugs than jsfunfuzz, and the two fuzzers were on a par with regard to the number of CVEs found: both found the same CVE-2018-0859. This result indicates that CodeAlchemist is more effective than the existing fuzzers in terms of its bug finding ability. In addition, jsfunfuzz found two bugs that are not discovered by CodeAlchemist, and CodeAlchemist found six bugs that are not discovered by jsfunfuzz. The result suggests that the two fuzzers could be complementarily used for finding bugs in JS engines.

We also note that three of the bugs CodeAlchemist found are still alive in the latest version of ChakraCore. The three bugs had been latent for about *seven* months undetected by other fuzzers or security researchers. This result reflects the fact that CodeAlchemist can find meaningful bugs deeply hidden in the JS engines.

2) Comparison on the Latest JS Engines: Now that we know CodeAlchemist can effectively find vulnerabilities on an old version of ChakraCore, it is natural to ask if CodeAlchemist is effective in finding vulnerabilities on the latest JS engines. We answer the question by running CodeAlchemist, jsfunfuzz, and IFuzzer on the latest stable version of the four major JS engines as of July 10th, 2018. In this experiment, we used the seeds we collected prior to the date (as we discussed in §VI-A), and the latest jsfunfuzz released before the date.

Table II presents the number of unique crashes each engine found after 2,112 CPU hours of fuzzing. In total, CodeAlchemist found 4.7× more unique bugs than jsfunfuzz: CodeAlchemist and jsfunfuzz found 14 and 3 unique bugs, respectively, while IFuzzer found nothing. CodeAlchemist was the only fuzzer who found bugs in three distinct JS engines: ChakraCore, V8, and JavaScriptCore. In contrast, jsfunfuzz was able to find bugs only in JavaScriptCore. One out of three bugs jsfunfuzz found was also found by CodeAlchemist. The other two bugs found by jsfunfuzz were not overlapped with the bugs found by CodeAlchemist. It is obvious from the results that CodeAlchemist prevails over the state-of-the-art JS engine fuzzers in both old and the latest version of JS engines.

TABLE II: The number of unique crashes found on the latest JS engines as of July 10th, 2018.

JS Engine	**CodeAlchemist**	**jsfunfuzz**	**IFuzzer**
ChakraCore 1.10.1	6	0	0
JavaScriptCore 2.20.3	6	3	0
V8 6.7.288.46	2	0	0
SpiderMonkey 61.0.1	0	0	0

```
var s0 = { // Var1
  get p() {},
  p : 2
};
function s1(s2) { // Func2
  ++s2.p;
}
Object.defineProperty(s0, 0, {}); // Expr3
s1(s0);  // Expr4
```

Fig. 10: A test case generated by CodeAlchemist triggering CVE-2018-8283 of ChakraCore 1.10.0.

E. Real-World Bugs Found

We have shown so far that CodeAlchemist is effective in finding bugs in JS engines within a controlled environment. However, we have also found numerous bugs in the latest JS engines. Particularly, we have run CodeAlchemist with a variety of different parameters for about a week on the same server machine we used. As a result, we found a total of 19 unique bugs on the four major JS engines.

Table III summarizes the list of bugs we have found so far. The third column of the table indicates whether the bug we found can trigger a crash in the corresponding browser. The fifth column indicates the security impact of each bug we found. We manually investigated each bug and mark them as "not exploitable" if we confirmed the root cause of the bug and it is obvious that the bug is not exploitable, e.g., a NULL dereference, and we also mark them as "likely exploitable" if a way the bug triggers memory corruption is similar to that of previously known PoC exploits.

First of all, we found 19 bugs including 11 exploitable bugs. Eight of them were not publicly known, although vendors had known them. The other 11 bugs we found were previously unknown. We reported all of them to the vendors and obtained 3 CVEs for the 7th, 8th, and the 9th bug.

We manually analyzed the other bugs too. As indicated by the fourth column of the table, all these bugs are related to diverse parts of the JS engines including JIT compilation, data parsing, and string handling. We believe this result highlights the impact of our research.

F. A Case Study

What do real-world bugs found by CodeAlchemist look? We now describe one of the bugs that we found in detail to answer the question. For responsible disclosure, we chose the one that is already patched (the 13th bug in Table III) as an example. Figure 10 shows a test case generated by CodeAlchemist, which triggers CVE-2018-8283 on ChakraCore 1.10.0. We simplified the test case for ease of explanation.

TABLE III: Unique bugs CodeAlchemist found.

Idx	JS Engine	Browser	Description	Impact	Status
1	JSC 2.20.3	Safari 11.1.1	Uninitialized memory access due to incorrect scoping	Exploitable	CVE-2018-4264
2	JSC 2.20.3	Safari 11.1.1	Use after free due to incorrect garbage collection	Exploitable	Confirmed
3	JSC 2.20.3	Safari 11.1.2	Memory corruption due to incorrect scoping	Exploitable	Confirmed
4	JSC 2.20.3	Safari 11.1.2	Memory corruption due to incorrect async function handling	Exploitable	Confirmed
5	JSC 2.20.3	Safari 11.1.2	Memory corruption due to incorrect regex parsing	Exploitable	Confirmed
6	JSC 2.20.3	Safari 11.1.2	Memory corruption due to incorrect date parsing	Exploitable	Confirmed
7	JSC 2.21.4 (beta)	Safari 11.1.2	Heap overflow due to incorrect string handling	Exploitable	CVE-2018-4437
8	JSC 2.21.4 (beta)	Safari 11.1.2	Memory corruption due to incorrect stack overflow handling	Exploitable	CVE-2018-4372
9	JSC 2.21.4 (beta)	Safari 12.0.0	Memory corruption due to incorrect JIT compilation	Exploitable	CVE-2018-4378
10	JSC 2.21.4 (beta)	Safari 11.1.2	Memory corruption due to incorrect string handling	Not Exploitable	Confirmed
11	V8 6.7.288.46	Chrome 67.0.3396.99	Out of bound access due to side effect in Float64Array	Exploitable	Confirmed
12	V8 6.7.288.46	Chrome 67.0.3396.99	Stack overflow due to incorrect recursively defined class handling	Not Exploitable	Confirmed
13	ChakraCore 1.10.0	-	Type confusion due to incorrect duplicated property handling	Exploitable	CVE-2018-8283
14	ChakraCore 1.10.1	-	Memory corruption due to incorrect yield handling in async function	Likely Exploitable	Reported
15	ChakraCore 1.10.1	-	Memory corruption due to incorrect JIT compilation	Likely Exploitable	Reported
16	ChakraCore 1.10.1	-	Use after free due to incorrect JIT compilation	Likely Exploitable	Reported
17	ChakraCore 1.10.1	Edge 43.17713.1000.0	Use after free due to incorrect JIT compilation	Not Exploitable	Confirmed
18	ChakraCore 1.10.1	Edge 43.17713.1000.0	Memory corruption due to incorrect JIT compilation	Not Exploitable	Confirmed
19	ChakraCore 1.10.1	Edge 43.17713.1000.0	Null dereference due to incorrect JIT compilation	Not Exploitable	Confirmed

The first statement declares an `Object` that has both (1) a property `p` and (2) the corresponding getter for the property. The second statement defines a function, and the third statement defines a property named "`0`" for the `Object s0`. After the property is defined, the JS engine traverses an internal dictionary that stores the mapping from a property name to a property object, and creates a new object that contains both the property `0` and `p`. At this point, the engine incorrectly judges the type of the property `p` as a 'getter' function type even though it is an integer value. Due to this incorrect type casting, a type confusion vulnerability triggers. Next, when the function `s1` is called, it accesses the property `p` of `s0`. However, due to the type confusion, this access tries to dereference an invalid getter function pointer, which causes a segmentation fault.

Note that generating such a code snippet is *not* trivial as each statement is deeply related to each other. The variable `s0` is used by both the function call statement and the `defineProperty` method. The function `s1` should be defined first and then be invoked with `s0` as its parameter. CodeAlchemist used the four code bricks annotated in Figure 10 to generate the test case: a code brick for the variable declaration statement (`Var1`), a code brick for the function declaration statement (`Func2`), and two code bricks for the expression statements (`Expr3, Expr4`). They have the following assembly constraints.

1) $\{\} \rightarrow$ `Var1` $\rightarrow \{$`s0: Object`$\}$
2) $\{\} \rightarrow$ `Func2` $\rightarrow \{$`s1: Function`$\}$
3) $\{$`s0: Object`$\} \rightarrow$ `Expr3` $\rightarrow \{$`s0: Object`$\}$
4) $\{$`s0: Object, s1: Function`$\}$
   $\rightarrow$ `Expr4` $\rightarrow \{$`s0: Object, s1: Function`$\}$

CodeAlchemist successfully assembled the code bricks based on their assembly constraints to generate the test case triggering the vulnerability.

VII. Discussion

Seed Selection. Semantics-aware assembly is essentially a seed-based fuzzing approach. Thus, collecting and selecting good seeds may substantially affect the performance of fuzzing as it does not create object types that are never seen in seeds. In our experiments, we obtained a set of seeds from existing JS test suites, but we believe CodeAlchemist can benefit by adopting state-of-the-art seed selection strategies [23], [25]. Additionally, automated seed generation techniques such as Skyfire [35] may help expand our seed pool.

Code Brick Selection. CodeAlchemist currently selects code bricks to be used next at random for every iteration. However, we believe that devising an intelligent code brick selection strategy can improve the fuzzing effectiveness. Although our current design choice is to follow a complete black-box approach as in LangFuzz [17], it is possible to employ a grey-box approach where the code brick selection is directed by code coverage or similar metrics. In addition, selecting the next code brick with probabilistic language models as in Skyfire [35] and TreeFuzz [24] can be effective in terms of bug finding. We believe this is an interesting future work.

Supporting Other Targets. It is straightforward to apply semantics-aware assembly to test other language interpreters or compilers because the core idea of our approach in assembling code bricks is indeed language agnostic. Furthermore, semantics-aware assembly can be more effective if it is used with statically-typed languages such as C and C++, because we can easily infer types of a variable without instrumenting the code. We leave it as future work to extend our algorithm to test compilers of other programming languages. We also believe we can apply our technique to find bugs in JS bindings [5], which input is allowed to contain JS code snippets, such as Node.js and PDF readers.

VIII. Related Work

A. Fuzzing

Fuzzing is a software testing technique for finding security vulnerabilities. It has been used by many security practitioners and researchers due to its significant practical impact. Fuzzing is typically divided into two major categories: mutation-based and generation-based fuzzing. Mutation-based fuzzers such as AFL [40] typically take in a set of seeds, and output test cases by mutating the seeds. There has been a surging interest on improving the effectiveness of mutation-based fuzzers [6], [25], [36]. On the other hand, generation-based fuzzers produce test cases based on a model, e.g., a grammar [17], [38]. IMF [14], for instance, automatically infers a model between system calls, and uses the model to produce a sequence of system calls to fuzz kernel code. CodeAlchemist is also in the category of generation-based fuzzing.

There are several previous generation-based fuzzers for testing interpreters and compilers. Most of them focus on generating syntactically valid programs based on a model to traverse deep in the interpreter (compiler). Since the very first thing that interpreters (compilers) do is to check whether a given program is syntactically valid, generating syntactically valid programs help traverse deep execution paths of the interpreter (compiler) under test.

There are several existing JS engine fuzzers that generate test cases based on pre-defined generation rules rather than relying on a complete language grammar. For example, domato [11] generates HTML, CSS, JS code to test DOM renderers of web browsers, and esfuzz [10] generates tests for ECMAScrirpt parsers. In addition, jsfunfuzz [27], which is a state-of-the-art JS engine fuzzer maintained by Mozilla, contains a huge number of generation rules that are manually built into the fuzzer.

Unfortunately, such fuzzers cannot create context-sensitive test cases by its design. Recall from §III, jsfunfuzz mainly focuses on syntactic, but not semantic validity of test cases it generates. It tries to heuristically mitigate runtime errors by variable renaming, but it still suffers from a high error rate based on our study. As such, finding new security vulnerabilities with jsfunfuzz is becoming difficult as our experimental results show.

In addition, Dewey *et al.* [8] address the problem of reducing runtime errors. Specifically, they propose a novel generation-based fuzzing algorithm that leverages constraint logic programming. The idea is to drive the test case generation towards specific language features with user-provided constraints. However, their approach requires an analyst to manually provide such constraints prior to fuzzing.

Several JS engine fuzzers try to generate test cases based on a set of given seeds. They fragmentize the seeds and reassemble the fragments to generate test cases. LangFuzz [17], the most successful JS engine fuzzer in this category, generates fragments by parsing down a given set of seeds into code fragments. It then mutates the seeds by replacing AST subtrees with the generated fragments. GramFuzz [13] and BlendFuzz [37] use the same intuition as LangFuzz, but they focus on other languages such as HTML, CSS, as well as JS. IFuzzer [33] improves upon LangFuzz by employing genetic programming to generate unseen JS test cases. TreeFuzz [24] and Skyfire [35] construct probabilistic language models from a given set of seeds in order to generate valid JS code snippets. None of the seed-based JS engine fuzzers deal with runtime errors while generating test cases.

B. JavaScript Analysis

JavaScript has become one of the most popular programming languages because of its flexibility, which enables programmers to write simple code fast. In contrast, the flexibility of the language raises the bar for traditional program analyses [26]. For example, the dynamic type system and the use of `eval` make analyzing JS painful as we discussed in §II-A. Thus, there has been much research on JS program analysis.

Dynamic instrumentation forms a basis for dynamic analyses, and there are several attempts that use dynamic instrumentation on JS code. Yu *et al.* [39] rewrite and instrument JS code to detect security policy violation. Sen *et al.* [28] present a dynamic analysis framework for JS, which provides a general-purpose dynamic JS code instrumentation mechanism. CodeAlchemist also employs the similar technique in order to obtain types of variables in code bricks.

Many researchers have built static type systems for JS starting from the seminal works by Anderson *et al.* [2] and Thiemann *et al.* [32]. Their approaches support only limited language features. Guha *et al.* [12] present λ_{JS}, a core language that embodies essential JS features including prototypes and first-class functions. Lerner *et al.* [18] propose a general framework for building JS type systems based upon λ_{JS}. Chandra *et al.* [7] handle a rich subset of JS, which can compute types for uninvoked functions. Our approach can benefit from the above approaches as we can make our assembly constraints more precise. We refer to recent survey papers [3], [29] for a more detailed overview of this area.

IX. Conclusion

We have presented CodeAlchemist, the first fuzzing system that generates semantically valid test cases for JS engines. CodeAlchemist learns the language semantics from a corpus of JS seed files, and generates a pool of code bricks that can be assembled later to construct semantically valid JS code snippets. We leverage both static and dynamic analysis technique to infer types of variables in each code brick, and use the information to build assembly constraints for each code brick, which help CodeAlchemist in judging which code bricks can be put together in which order. This simple approach significantly reduced runtime errors for JS engine fuzzing while producing highly-constructed test cases. CodeAlchemist found 19 bugs in the four major JS engines. We have reported all our findings to the vendors.

Acknowledgement

We thank our shepherd, Deian Stefan, and the anonymous reviewers for their helpful feedback. We also would like to thank Jaeseung Choi for fruitful discussion. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.B0717- 16-0109, Building a Platform for Automated Reverse Engineering and Vulnerability Detection with Binary Code Analysis).

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[2] C. Anderson, P. Giannini, and S. Drossopoulou, "Towards type inference for javascript," in *Proceedings of the ACM European Conference on Object-Oriented Programming*, 2005, pp. 428–452.

[3] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, "A survey of dynamic analysis and test generation for JavaScript," *ACM Computing Surveys*, vol. 50, no. 5, pp. 66:1–66:36, 2017.

[4] Apple Inc., "WTFCrash in JavaScriptCore," https://github.com/WebKit/webkit/blob/82bae82cf0f329dbe21059ef0986c4e92fea4ba6/Source/WTF/wtf/Assertions.cpp#L261.

[5] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan, "Finding and preventing bugs in javascript bindings," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2017, pp. 559–578.

[6] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2015, pp. 725–741.

[7] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi, "Type inference for static compilation of javascript," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2016, pp. 410–429.

[8] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *Proceedings of the International Conference on Automated Software Engineering*, 2014, pp. 725–730.

[9] Ecma International, "ECMAScript 2015 language specification," https://www.ecma-international.org/ecma-262/6.0/, 2015.

[10] M. Ficarra and B. Zhang, "esfuzz," https://github.com/estools/esfuzz.

[11] I. Fratric, "domato," https://github.com/google/domato.

[12] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of JavaScript," in *Proceedings of the ACM European Conference on Object-Oriented Programming*, 2010, pp. 126–150.

[13] T. Guo, P. Zhang, X. Wang, and Q. Wei, "GramFuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation," in *Proceedings of the International Conference on Informatics Applications*, 2013, pp. 212–215.

[14] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 2345–2358.

[15] A. Hidayat, "Esprima," http://esprima.org/.

[16] C. Holler, "Security testing at Mozilla," http://issta2016.cispa.saarland/security-testing-at-mozilla/, keynote at ISSTA 2016.

[17] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the USENIX Security Symposium*, 2012, pp. 445–458.

[18] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi, "TeJaS: Retrofitting type systems for JavaScript," in *Proceedings of the Symposium on Dynamic Languages*, 2013, pp. 1–16.

[19] H. Li and J. Tang, "Cross the wall: Bypass all modern mitigations of Microsoft Edge," in *Proceedings of the Black Hat Asia*, 2017.

[20] M. Molinyawe, A.-A. Hariri, and J. Spelman, "$hell on Earth: From browser to system compromise," in *Proceedings of the Black Hat USA*, 2016.

[21] Mozilla, "Inheritance and the prototype chain," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain.

[22] Node.js Foundation, "Node.js," https://nodejs.org.

[23] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS fuzzer seed selection with trace distillation," in *Proceedings of the USENIX Security Symposium*, 2018, pp. 729–743.

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

[25] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the USENIX Security Symposium*, 2014, pp. 861–875.

[26] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2010, pp. 1–12.

[27] M. Security, "funfuzz," https://github.com/MozillaSecurity/funfuzz.

[28] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for JavaScript," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2013, pp. 488–498.

[29] K. Sun and S. Ryu, "Analysis of JavaScript programs: Challenges and research trends," *ACM Computing Surveys*, vol. 50, no. 4, pp. 59:1–59:34, 2017.

[30] Y. Suzuki, "escodegen," https://github.com/estools/escodegen.

[31] TC39, "Test262: ECMAScript conformance test suite," https://github.com/tc39/test262.

[32] P. Thiemann, "Towards a type system for analyzing javascript programs," in *Proceedings of the ACM European Conference on Programming Languages and Systems*, 2005, pp. 408–422.

[33] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "IFuzzer: An evolutionary interpreter fuzzer using genetic programming," in *Proceedings of the European Symposium on Research in Computer Security*, 2016, pp. 581–601.

[34] W^{3}Techs, "Usage statistics of javascript for websites, July 2018," https://w3techs.com/technologies/details/cp-javascript/all/all.

[35] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2017, pp. 579–594.

[36] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2013, pp. 511–522.

[37] D. Yang, Y. Zhang, and Q. Liu, "BlendFuzz: A model-based framework for fuzz testing programs with grammatical inputs," in *Proceedings of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2012, pp. 1070–1076.

[38] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.

[39] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript instrumentation for browser security," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 2007, pp. 237–249.

[40] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.