Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases

Sung Ta Dinh*, Haehyun Cho*, Kyle Martin†, Adam Oest‡, Kyle Zeng*, Alexandros Kapravelos†, Gail-Joon Ahn*§, Tiffany Bao*, Ruoyu Wang*, Adam Doupé*, and Yan Shoshitaishvili*
*Arizona State University, †North Carolina State University, ‡PayPal, Inc., §Samsung Research
*{tdsung, haehyun, zengyhkyle, gahn, tbao, fishw, doupe, yans}@asu.edu
†{kdmarti2, akaprav}@ncsu.edu, ‡{aoest}@paypal.com

***Abstract*—JavaScript runtime systems include some specialized programming interfaces, called binding layers. Binding layers translate data representations between JavaScript and unsafe low-level languages, such as C and C++, by converting data between different types. Due to the wide adoption of JavaScript (and JavaScript engines) in the entire computing ecosystem, discovering bugs in JavaScript binding layers is critical. Nonetheless, existing JavaScript fuzzers cannot adequately fuzz binding layers due to two major challenges: Generating syntactically and semantically correct test cases and reducing the size of the input space for fuzzing.**

In this paper, we propose Favocado, a novel fuzzing approach that focuses on fuzzing binding layers of JavaScript runtime systems. Favocado can generate syntactically and semantically correct JavaScript test cases through the use of extracted semantic information and careful maintaining of execution states. This way, test cases that Favocado generates do not raise unintended runtime exceptions, which substantially increases the chance of triggering binding code. Additionally, exploiting a unique feature (relative isolation) of binding layers, Favocado significantly reduces the size of the fuzzing input space by splitting DOM objects into equivalence classes and focusing fuzzing within each equivalence class. We demonstrate the effectiveness of Favocado in our experiments and show that Favocado outperforms a state-of-the-art DOM fuzzer. Finally, during the evaluation, we find 61 previously unknown bugs in four JavaScript runtime systems (Adobe Acrobat Reader, Foxit PDF Reader, Chromium, and WebKit). 33 of these bugs are security vulnerabilities.

I. Introduction

The use of JavaScript has expanded beyond web browsers into the entire computing ecosystem as a general-purpose programming language. As a result, JavaScript engines are embedded in a variety of commercial software (*e.g.*, Adobe Acrobat and Node.js). JavaScript engines often provide important functionality through a binding layer, which is usually implemented in unsafe languages such as C and C++. While the JavaScript engines are being heavily studied, fuzzed, and hardened, their binding layers are frequently overlooked. This is exemplified by the introduction of multiple JavaScript fuzzers over the past few years, none of which could be used to fuzz binding code in non-browser environments [24, 27, 29, 34, 37, 40, 41, 55, 56]. However, due to the complexity in the implementation of binding layers in JavaScript engines, vulnerabilities in these layers are not rare [11]. Therefore, there is a pressing need to design JavaScript fuzzers to efficiently fuzz JavaScript code and effectively find bugs in these binding layers.

Even without considering the binding layers, it is difficult to effectively fuzz JavaScript engines in the first place. Researchers found that for fuzzing JavaScript engines, the quality of the initial fuzzing input (*i.e.*, seeds) greatly impacts the fuzzing performance [44]. This is because JavaScript engines do not directly consume the user-provided JavaScript code. These engines will parse user input into an abstract syntax tree (AST) and then process the tree. User inputs that cannot be transformed into an AST are easily rejected. Hence, JavaScript test cases generated by fuzzers that are unaware of JavaScript specifications are likely to be malformed and rejected before being processed.

To generate syntactically correct JavaScript code as test cases, modern JavaScript engine fuzzers use context-free grammars [8, 24, 29, 37, 57] or existing semantically correct test cases [27, 40, 55, 56]. However, only being syntactically correct is not enough for JavaScript engines to process a test case, as many JavaScript statements have interdependent relationships. Failing to capture such relationships will lead to generating semantically incorrect code that raises runtime exceptions when being processed. While no JavaScript fuzzers generate fully semantically correct code as test cases, some fuzzers can generate test cases in a semantic-aware manner [27, 40, 56]. However, the percentage of rejected test cases that are generated by these semantic-aware fuzzers is still a significant problem.

Unfortunately, existing fuzzers are likely to have a difficult time generating test cases that can adequately fuzz JavaScript binding layers. As shown in Listing 1, a typical JavaScript test case that triggers the execution of binding code *once* involves at least two steps: (1) Creating the object and (2) setting a property of the object or calling a function of the object. Due to the excessive number of JavaScript exceptions that randomly generated test cases raise, it is practically impossible for existing fuzzers to generate legitimate JavaScript code that

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23xxx
www.ndss-symposium.org

```
var cb = this.getField("CheckBox");
cb.checkThisBox(0,true);
```

Listing 1: An example JavaScript test case that triggers the execution of binding code to check a `checkbox`.

covers both steps. Not to mention, generating a sequence of such snippets to execute binding code multiple times.

Another challenge for effectively fuzzing the binding layer is the enormous input space. There are many object types that are accessible with JavaScript through the binding layer as a Document Object Model (DOM) (*e.g.*, in Chromium, there are more than 1,000 DOM binding objects). Each DOM object may have a multitude of methods and properties, some of which may require hard-to-satisfy arguments such as other DOM objects. Creating all objects to enumerate all properties and manipulate all methods is simply infeasible. An effective fuzzer that adequately fuzzes JavaScript binding code should be aware of the unique features of this layer when generating test cases. With this embedded awareness built in, a fuzzer can optimize test case generation by reducing the size of the input space.

One unique feature of the JavaScript binding layer is the relative *isolation* of different DOM objects. Intuitively, different DOM objects (*e.g.*, for Adobe Acrobat, `spell.check()` in its spell module and `Net.HTTP.request()` in its Net.HTTP module) in the binding layer are implemented as separate native modules, unless an object defined in one module can be used by code in another module. A JavaScript test case that calls `spell.check()` before `Net.HTTP.request()` is essentially equivalent to another test case that calls the two methods in reverse order. We may define a *DOM objects relation* where an object can use another object as a value to its properties or a parameter to its methods. Based on the relations between DOM objects, we may divide the entire input space into equivalence classes. In our example, `spell.check()` and `Net.HTTP.request()` will fall into different equivalence classes. Object-relation-aware fuzzers may only mutate DOM objects within each equivalence class. This greatly reduces the size of the input space. Existing JavaScript fuzzers are not aware of the isolation of different DOM objects, which makes them unsuitable for adequately fuzzing the JavaScript binding layer.

In this paper, we propose a novel fuzzing approach, codenamed Favocado, that focuses on finding vulnerabilities in the binding layers of JavaScript engines. Favocado tackles the aforementioned challenges by (1) generating syntactically and semantically correct test cases to eliminate runtime exceptions and (2) generating object-relation-aware test cases to significantly reduce the large input space.

Generating semantically correct test cases. Favocado collects semantic information of binding interfaces by analyzing existing JavaScript API references and Interface Definition Language (IDL) files. IDL files define interfaces within binding code that are accessible to JavaScript. Additionally, Favocado manages states during mutation to ensure the following correctness in semantics: All generated JavaScript statements may only access objects, properties, and methods that are currently available (*e.g.*, Favocado is able to generate JavaScript statements that do not access previously deallocated objects).

Reducing the size of the input space. Favocado excavates relations between binding objects from the collected semantic information. Then it separates all binding objects into multiple equivalence classes based on their relations. Finally, Favocado focuses on fuzzing JavaScript binding layer by each equivalence class.

To demonstrate the generality and effectiveness of Favocado, we thoroughly evaluate our prototype with different types of binding objects (PDF, Mojo, and DOM). These binding objects are implemented in four different JavaScript runtime systems (Adobe Acrobat Reader, Foxit PDF Reader, Chromium, and WebKit). During our evaluation, Favocado finds 61 previously unknown bugs, which includes 33 severe security vulnerabilities. Our evaluation results show the effectiveness of Favocado, which outperforms the state-of-the-art DOM fuzzer, Domato.

Contributions. This paper makes the following contributions:

- We propose Favocado, a novel approach for fuzzing binding layers of JavaScript engines. Favocado generates semantically correct JavaScript test cases based on extracted semantic information and tracking states mutation. Favocado also reduces the input space by being aware of relations between DOM objects.
- We implement a prototype of Favocado and thoroughly evaluate it against real-world binding code in four different JavaScript runtime systems (Adobe Acrobat Reader, Foxit PDF Reader, Chromium, and WebKit) to demonstrate the effectiveness of Favocado. We also compare Favocado against Domato and show that Favocado outperforms Domato.
- We responsibly analyzed and disclosed all bugs found by Favocado that include 33 security vulnerabilities. By the time of writing, 13 bugs have been assigned CVE entries during the responsible disclosure process.

To foster further research, we open source the prototype of Favocado that we developed as part of our research. The repository is at https://github.com/favocado/Favocado.

II. Background

In this section, we present the background of the JavaScript binding code fuzzing problem. We will introduce JavaScript binding code, the terms used in the paper, as well as previous related work on fuzzing JavaScript engines and the binding code.

A. Binding Code

JavaScript is a dynamic high-level programming language interpreted by JavaScript engines (*e.g.*, Chrome V8, SpiderMonkey, and Chakra). Currently, the use of JavaScript is not limited in implementing interactive web pages; it is also used as a general-purpose programming language in both browsers and many software systems (*e.g.*, Adobe Acrobat Reader). For example, applications such as Adobe Acrobat, Blink, and PDFium utilize JavaScript engines to provide dynamic/interactive content through JavaScript code embedded in PDF documents. Because JavaScript cannot be used to directly implement low-level functionalities (*e.g.*, memory management

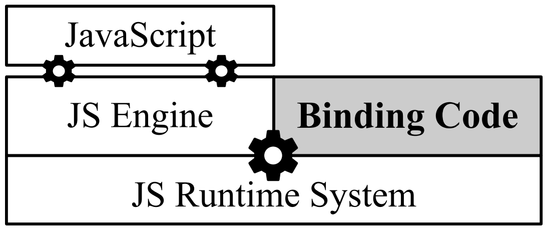


Fig. 1: Binding code is used to extend functionalities of JavaScript by translating data representations between JavaScript and native code.

and file access), those functions are implemented in unsafe languages (*e.g.*, C and C++) in JavaScript engines to enable the extensive use.

To use such additional functionalities, JavaScript runtime systems have *binding code*, which is a native component of JavaScript engines as shown in Figure 1. Binding code translates data representations: It creates and maps necessary data types between JavaScript and native code. Native functions implemented in binding code provide JavaScript objects by defining them through an interface definition language (IDL). When JavaScript creates such objects, binding code dynamically generates corresponding native data formats and types them with JavaScript variables. Then scripts can call native functions or control data of native components. For example, in browsers, the DOM is a programming interface for controlling HTML documents (web pages). The DOM provides DOM objects as a programming interface so that JavaScript can easily manipulate the structures and styles of a document and its elements. On the other hand, the DOM internally implements logical data structures of a document and functions in a low-level language that defines how a document can be accessed and changed by JavaScript. However, unfortunately, during the translation process, type-, and memory-safety features of JavaScript cannot be interpreted. Binding code is implemented in low-level unsafe languages and vulnerabilities are not rare [11].

A vulnerability in binding code can be a serious security threat because "JavaScript is everywhere." As an example, PDFium is a PDF document rendering module of Chrome browser and Foxit PDF reader, which is bound with V8 engine to support customizing PDF documents through JavaScript code embedded in PDFs. If there exists an exploitable JavaScript binding bug within a PDF reader, then not only is the standalone application vulnerable but also web browsers and other PDF readers that include this binding code as a component. Therefore, binding code must execute the translation process with rigorous principles to prevent security violations which frequently happens and makes JavaScript code exploitable [11].

To detect security flaws in binding layers, many research works have been proposed [11, 12, 22, 33, 35, 36, 53, 54]. They have focused on bugs that occur when binding code omits necessary checks, violates data translation rules, and mishandles exceptions through various static analysis approaches and a dynamic bug detector. However, these solutions have limited applications and scopes, and thus, are limited in finding vulnerabilities only within their scopes. On the other hand, albeit fuzzing has been proven practical for discovering vulnerabilities in software, existing JavaScript fuzzers are not practical in terms of fuzzing binding code.

B. Terminology

Throughout this paper, we use the term *semantics/semantic information of binding code* to refer to static type signatures of all methods and objects in binding code that JavaScript can access to. In addition, we use the term *semantically correct test case* to refer to JavaScript statements that use (1) correct semantic information of binding code and (2) valid language semantics in the execution context (*i.e.*, correctly using previously defined variables based on their types). For example, the Line 2 of Listing 1 is semantically correct because the `cb` variable is pointing to the `CheckBox` type object that has the `checkThisBox` method. Also, it uses the correct types of arguments for calling the method.

C. Fuzzing JavaScript Engines

Due to the high severity of vulnerabilities in JavaScript runtime environments, a number of research work have attempted to fuzz JavaScript engines for finding vulnerabilities [24, 27, 29, 34, 37, 40, 41, 55, 56]. For fuzzing JavaScript engines, most research work have focused on generating *syntactically* valid JavaScript test cases [24, 29, 37, 41, 55]. Despite their successful fuzzing efforts, they did not consider JavaScript semantics, and thus, could not generate test cases effectively—many test cases merely end up as JavaScript runtime errors [27, 56]. Skyfire [56] was proposed to generate test cases through the probabilistic context-sensitive grammar that defines syntax features and semantic rules by learning from existing samples. CodeAlchemist [27] was proposed to generate semantically-aware JavaScript code by using small code blocks collected from a large corpus. Unfortunately, there is no JavaScript engine fuzzer that can generate *semantically correct* test cases all the time.

Recently proposed JavaScript engine fuzzers tried to reduce the input space. DIE [40] has proposed two mutation strategies, structure-preserving mutation and type-preserving mutation. They, also, reduce the input space by utilizing known proof of concept (PoC) exploits or existing test cases. Montage [34] showed its outperformed efficacy by leveraging a neural network language model (NNLM) to generate test cases based on code fragments of previously reported vulnerabilities, similar to DIE. Their (DIE and Montage) design choice allowed them to overcome the fundamental limitation of other JavaScript engine fuzzers: simply producing generic test cases is not really effective to find vulnerabilities in JavaScript engines because of the huge search space.

D. Fuzzing the Binding Code of JavaScript Runtime Systems

While many research projects have been attempting to fuzz the core of JavaScript engines for discovering vulnerabilities, the binding code area of JavaScript engines has not been extensively explored yet in the context of fuzzing. A couple of fuzzers such as Domato and DOMFuzz (deprecated) have been used for fuzzing the Document Object Model (DOM) which is a widely used programming interface (binding code) in browsers for accessing a document on the web from JavaScript [21, 45].

Domato revealed some severe vulnerabilities from DOM objects of web browsers. However, it relied on manual development of a *grammar*, detail specifications for invoking each method and assigning objects to properties of each object, to generate test cases. Therefore, Domato cannot avoid huge manual efforts to create a grammar file for fuzzing each binding code (they implemented a grammar file for fuzzing DOM objects with 5.6K+ LoC). Additionally, Domato lacks the ability to generate semantically correct test cases. We need a binding code fuzzer that can perform semantically correct fuzzing and can minimize manual efforts for extracting complete semantics of targeted binding code so that it can be used for fuzzing any binding code in a general manner.

E. JavaScript Engine Fuzzers for Binding Code.

JavaScript engine fuzzers such as DIE [40] and Montage [34] utilize PoCs of known vulnerabilities and regression test cases because such PoCs or test suite are specially designed for exploiting specific features of a JavaScript engine. Therefore, by utilizing them, fuzzers can effectively generate test cases for exploring distinct and complex execution paths. However, for finding bugs in binding code, we need to test a series of various JavaScript statements setting properties and invoking APIs with correct semantics, which does not require complex syntactics or code patterns learned from test suites. Thus, generating test cases syntactically similar to such existing PoC exploits is not an effective way to find vulnerabilities in binding code.

CodeAlchemist [27], a state-of-the-art semantics-aware fuzzer, proposed an approach to create *more* semantically valid test cases. However, their evaluation results showed that only less than 20% of test cases which have more than 5 statements executed without raising any runtime error [27]. We provide experimental results regarding the availability of CodeAlchemist as a binding code fuzzer in Section V-B.

On the other hand, mutational fuzzers using context-free grammars such as Superion [57] and Nautilus [8] requires well-developed grammars. Therefore, those fuzzers have the same limitations as the existing binding code fuzzers.

III. Overview

Fuzzing JavaScript engine and binding code faces two major challenges: (1) generating semantically correct test cases, and (2) reducing the size of the input space. These challenges are not unique to fuzzing JavaScript engines. In fact, they are prevalent in fuzzers for programs that take highly structured test cases as input. In this section, we first describe these challenges in detail for fuzzing binding code. We then discuss why they must be tackled (Section III-A) and provide a high-level overview of how Favocado addresses these challenges (Section III-B).

A. Requirements for Fuzzing Binding Code

1) Semantically Correct Test Cases: First of all, for fuzzing binding code of JavaScript engines, we should have complete *semantics of targeted binding code*. Fuzzing binding code with JavaScript code requires allocating (define) a binding object, assigning a specific value to a property, and calling a method with correct-type arguments. Therefore, without accurate semantic information of targeted binding code, we cannot generate test cases for fuzzing binding code. Also, we need an automated approach that constructs semantic information of targeted binding code to avoid manual development.

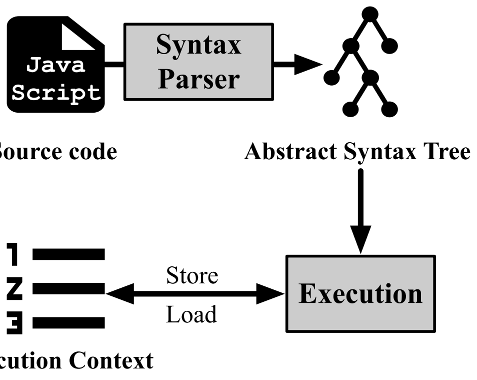
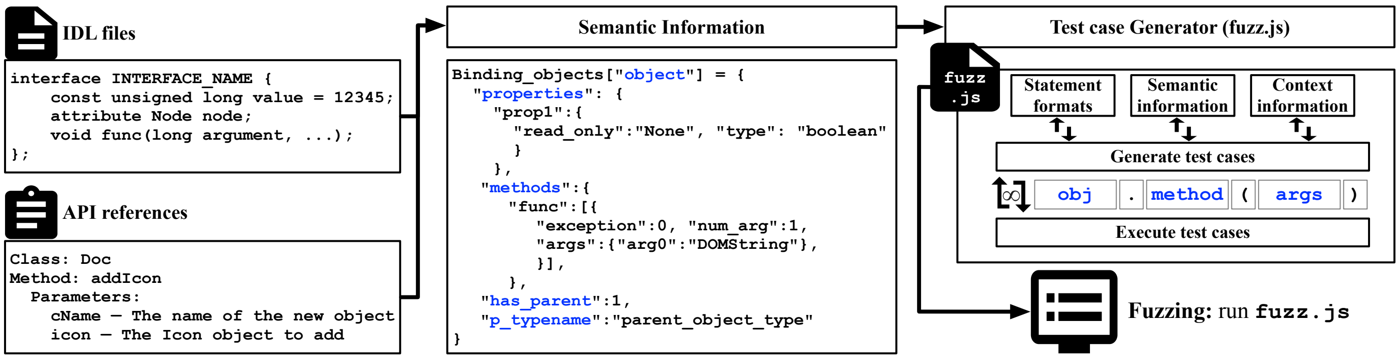


Fig. 2: JavaScript engines first parse source code to an AST. Then, they execute it, managing the execution context information dynamically.

Next, generating syntactically valid test cases is not only a prerequisite but we also need to generate semantically correct test cases for maximizing the effectiveness of fuzzing against binding code. We note that semantically correct test cases stand for JavaScript statements that have valid semantics of targeted binding code (*e.g.*, correct use of method names, argument types, and return type of binding code) as well as valid runtime semantics (*e.g.*, correct use of previously defined variables and objects based on their types). Invalid test cases, which cause runtime errors (syntax, reference, type, and range errors) in the middle of execution, seriously impede the progress of fuzzing. Unfortunately, state-of-the-art JavaScript engine fuzzers introduce high error rates of test cases [27, 34, 40, 45]. Also, Domato, a binding code fuzzer leveraging context-free grammars, cannot consistently generate semantically correct test cases [21].

Figure 2 illustrates how a JavaScript engine executes JavaScript code. To execute JavaScript code, JavaScript engines first generate an Abstract Syntax Tree (AST) by parsing the source code through a syntax parser. If the syntax of the code is not correct, it will not generate AST or execute the source code. Then, the JavaScript engine starts to execute the code within the execution context that controls the scope of the code and contains up-to-date information of the current program state. The JavaScript engine dynamically manages the execution context. If a statement accesses a variable that is out of the scope or an object that has been deallocated, the JavaScript engine raises a runtime error and stops executing the code.

We should generate and execute JavaScript statements including method calls for fuzzing binding code because each bug in binding code can only be discovered by executing a series of JavaScript statements accessing binding objects. Hence, a fuzzer needs to input many JavaScript statements



```
interface INTERFACE_NAME {
    const unsigned long value = 12345;
    attribute Node node;
    void func(long argument, ...);
};
```

```
Class: Doc
Method: addIcon
  Parameters:
    cName — The name of the new object
    icon — The Icon object to add
```

```
Binding_objects["object"] = {
  "properties": {
    "prop1":{
      "read_only":"None", "type": "boolean"
      }
    },
   "methods":{
     "func":[{
       "exception":0, "num_arg":1,
       "args":{"arg0":"DOMString"},
       }],
     },
   "has_parent":1,
   "p_typename":"parent_object_type"
}
```

Fig. 3: The overview of Favocado. After extracting semantic information of binding objects, Favocado starts to fuzz binding code inside the target JavaScript runtime system with syntactically and semantically correct test cases.

at once as a basic testing unit for fuzzing. If a test case contains a semantically incorrect statement, the test case has to stop executing and retire—fuzzers cannot evaluate the test case where security vulnerabilities may exist because of such invalid JavaScript statements. Consequently, to achieve highly effective fuzzing on JavaScript binding code, it is critical to prevent runtime errors by carefully generating semantically correct JavaScript statements that do not transgress the execution context.

2) Reducing the Size of the Input Space: A large input space severely hinders the effectiveness of fuzzing. Thus, we need to reduce the size of the input space of binding code [34, 40]. Recent JavaScript engine fuzzers utilize existing test cases or PoC exploits of previous vulnerabilities to learn their syntax so that they can reduce the input space and quickly traverse complex execution paths [27, 34, 40]. The input space of binding code is huge, but our goal is not only covering deep execution paths. When fuzzing binding code, we need to focus on generating various method call sequences and changing arguments and properties of binding objects, which is more important than exploring deep execution paths. Therefore, we do not leverage an existing test suite or PoCs of previous vulnerabilities for reducing the input space similar to the recent JavaScript engine fuzzers.

B. Our Approach

Our goal is to achieve the two requirements discussed in Section III-A.

Favocado first parses semantic information from the Interface Definition Language (IDL) files or API references. For constructing complete semantic information of targeted binding code, Favocado parses IDL files when source code is available. If source code is not publicly opened, we need to use API reference manuals (*e.g.*, JavaScript for Acrobat API reference [7]). By parsing IDL files or API references, Favocado obtains semantic information of binding objects (their methods with arguments, and their properties), which include exact types and possible values (discussed in Section IV-A). Extracted semantic information can directly be used for generating test cases.

Next, Favocado generates a JavaScript file (a test case generator) with the semantic information of binding code and starts fuzzing by executing the test case generator on a target JavaScript runtime system. The test case generator randomly selects a JavaScript statement format among predefined statement formats that include statements defining objects, calling methods, assigning values to properties, and so forth as shown in Figure 5. It, then, uses the semantic information of binding code and context information of a fuzzing process (*i.e.*, the runtime semantics of test cases such as a list of previously defined variables with their types) to complete a statement, preventing unexpected runtime errors. As an example, when Favocado constructs a statement such as `car.drive(man)`, Favocado checks whether the `man` object has been properly defined or not. In addition, even though Favocado knows that the object has been previously defined, it checks whether the object is still alive and accessible. This is because, for example, runtime errors can occur when a statement accesses an object after invoking a method that deallocates the object such as `removechild`. If the object was deallocated by such methods, Favocado creates the object again and executes the constructed statement, thus avoiding a runtime error. It also remembers the variable name pointing to the newly defined object for later use. We discuss our test case generation mechanism in detail in Section IV-B.

Furthermore, Favocado divides an input space to increase the effectiveness of fuzzing binding code. Specifically, to deal with the large input space, Favocado randomly selects several binding objects for a single fuzzing process to focus only on them and runs multiple fuzzing instances concurrently. When Favocado selects binding objects, it considers relationships of objects so that objects related to each other can be fuzzed together (discussed in Section IV-B).

In summary, Favocado can fuzz any type of binding code in a general manner with syntactically and semantically correct test cases, preventing runtime errors (not wasting any test case, except for when it is intentionally generating erroneous statements for finding bugs). It is worth noting that the design of Favocado is not limited to fuzzing binding code of JavaScript engines. We believe our approach can also be used for fuzzing binding code of the other scripting languages.

IV. Design

In this section, we describe the design of Favocado. Favocado consists of two parts: (1) semantic information construction (Section IV-A) and (2) dynamic error-safe test case generator (Section IV-B). Figure 3 illustrates the overview of Favocado design: After extracting semantic information of binding objects, it starts fuzzing binding code inside a target runtime system, dynamically generating syntactically and semantically correct test cases.

A. Semantic Information Construction

Favocado relies on the semantic information extracted from either IDL files or JavaScript API references of software systems. Specifically, Favocado parses the following information.

(1) Binding object names. Favocado records a name of each object, and a name of a parent object if the object has a parent to check whether an object has inherited methods or properties.

(2) Binding object methods. Favocado obtains each method's name and all arguments' exact types to generate a valid method call statement. Also, Favocado checks whether or not a method can raise an exception to handle it. A return type of a method is decided as well.

(3) Binding object properties. Favocado parses a name, type, and possible string values (if a type of property is a string) of each property. Furthermore, it checks whether a property is read-only.

```
Binding_objects["HTMLDialogElement"] = {
  "properties":
  {
    "open":
    {
      "read_only":"None", "type":"boolean"
    },
    "returnValue":
    {
      "read_only":"None", "type":"DOMString"
    }
  },
  "methods":
  {
    "close":
    {
      "exception":0, "numarg":1,
      "args":{"arg0":"DOMString"},
    },
    "showModal":
    {
      "exception":1, "numarg":0,
      "args":{},
    },
    "show":
    {
      "exception":0, "numarg":0,
      "args":{},
    }
  },
  "has_parent":1,
  "p_typename":"HTMLElement"
}
```

Listing 2: An example of extracted semantic information of the HTMLDialogElement object. Favocado generates a test case while fuzzing binding code by using this semantic information.

Listing 2 shows an example of extracted semantic information. The `HTMLDialogElement` object has two properties and three methods. Also, this object has a parent object: `HTMLElement`. Types of the two properties are `boolean` and `DOMString`. The `close` method has one argument of which type is `DOMString`. Additionally, we notice that the `showModal` method does not have an argument, but it throws an exception if a certain condition is unmet. Therefore, Favocado should dynamically handle this exception through the `try–catch` statement. Otherwise, the fuzzing process can stop because of the exception. This semantic information is used directly to generate a test case while fuzzing.

Before fuzzing, Favocado finds binding objects related each other using type information of method arguments, return values, and properties. To this end, Favocado analyzes the extracted semantic information. If an object is used as an argument of a method, a return type, or a type of a property in another object, Favocado records them so that they can be fuzzed together (discussed in Section IV-B). An example of related binding objects is shown in Listing 3. Favocado discovered the `ImageCapture` object is related to `Blob`, `ImageBitmap`, `MediaStreamTrack`, and `PhotoCapabilities` objects because these objects are return types of the `ImageCapture` object's methods. Also, the `Crypto` object has `ArrayBufferView` and `SubtleCrypto` objects as related objects because they are used as an argument of a method and a type of a property, respectively.

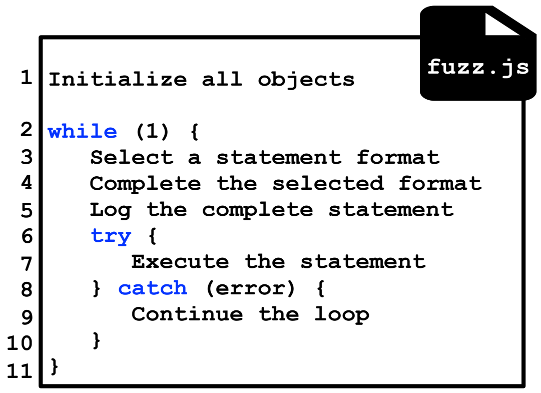
```
"ImageCapture":
[{
  "Blob", "ImageBitmap", "MediaStreamTrack", "
      PhotoCapabilities"
}]

"Crypto":
[{
  "ArrayBufferView", "SubtleCrypto"
}]
```

Listing 3: An example of related objects discovered by Favocado.

B. Dynamic Test Case Generator

The test case generator is a JavaScript file that dynamically generates and executes JavaScript statements inside a target system. This design enables us to actively handle runtime errors while fuzzing so that we can perform fuzzing binding code without losing a generated test case. 40% of test cases generated by state-of-the-art JavaScript engine fuzzers may lead to runtime errors, which prevent fuzzers from exploring corresponding execution paths where vulnerabilities may exist.

Dividing an input space into equivalence classes. In fuzzing, a large input space not only increases the time to generate a test case, but also hinders fuzzer's capability of exploring deep execution paths [34, 40]. This also applies to fuzzing JavaScript binding code. There are numerous DOM objects in each JavaScript runtime system. For example, in Chromium, there are more than 1,000 DOM objects. Each DOM object may have a multitude of methods and properties, some of which may require hard-to-satisfy arguments such as other



```
Initialize all objects

while (1) {
   Select a statement format
   Complete the selected format
   Log the complete statement
   try {
      Execute the statement
   } catch (error) {
      Continue the loop
   }
}
```

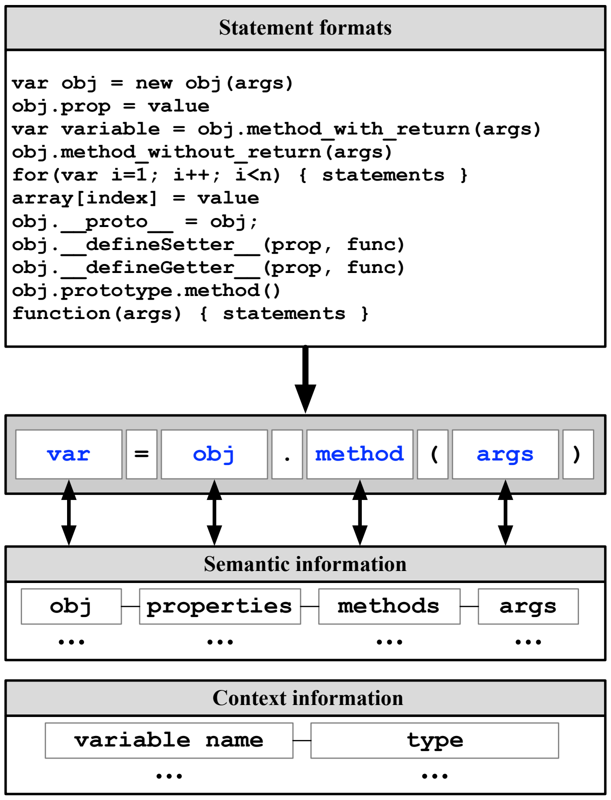
Fig. 4: The execution flow of the test case generator. After initializing all objects targeted in a test case generator, Favocado starts fuzzing targeted binding objects.

DOM objects. Therefore, fuzzing all combinations of all DOM objects (and their properties and methods) is simply infeasible.

We notice a unique feature in JavaScript binding layers: the *isolation* among DOM objects. Different DOM objects (e.g., for Adobe Acrobat, `spell.check()` in the spell module and `Net.HTTP.request()` in the Net.HTTP module) in the binding layer are implemented as separate native modules. A test case that invokes `spell.check()` before `Net.HTTP.request()` is equivalent to another test case that invokes the two methods in reverse order. We define a *DOM objects relation* where an object uses another object as a value to its properties, a parameter to its methods, or a return type of its methods. For example, Listing 3 shows that `Crypto` object is related to `ArrayBufferView` (a return type of `getRandomValues` method in `Crypto`) and `SubtleCrypto` (a property of `Crypto`). `ImageCapture` object has four related objects. Based on the relation between DOM objects, we may divide the entire input space into equivalence classes. Favocado then only mutate and fuzz DOM objects within each equivalence class. This way, the size of the input space is significantly reduced.

Favocado analyzes the extracted semantic information to identify the relationships of binding objects. Once fuzzing started, Favocado randomly selects binding objects based on the analysis result and generates test cases for them.

Test case generator. Favocado creates the test case generator (`fuzz.js`) that executes on a JavaScript runtime system, fuzzing binding objects. In the test case generator, the semantic information of binding objects, statement formats, and predefined JavaScript statements for initializing binding objects are included. We could not completely automate generating the statements for initializing some binding objects in each JavaScript runtime system. This is because many objects can be initialized easily without difficulty, but some binding objects require environment-specific data such as IP addresses, specific data formats such as image files, or have dependencies on other objects to initialize them, which cannot be generated from the semantic information and thus cannot be automated. We note that Domato [21] also manually implemented DOM objects initialization statements for fuzzing. To balance fuzzing performance and engineering efforts in this paper, we leave the



```
var obj = new obj(args)
obj.prop = value
var variable = obj.method_with_return(args)
obj.method_without_return(args)
for(var i=1; i++; i<n) { statements }
array[index] = value
obj.__proto__ = obj;
obj.__defineSetter__(prop, func)
obj.__defineGetter__(prop, func)
obj.prototype.method()
function(args) { statements }
```

Fig. 5: An example of generating a test case. Favocado selects a statement format and completes it by leveraging the semantic information of targeted binding code. Also, it manages the context information to avoid unexpected runtime errors.

automated binding object initialization as future work. In addition, for providing context information of a fuzzing process, there is a list of allocated variables in a test case generator. This data structure holds information on available variable names with their types. A test case generator dynamically maintains the context information to prevent unexpected runtime errors such as reference and type errors (except for intentionally generating erroneous statements, for example, where Favocado tries to trigger bugs by using different types).

Favocado executes a test case generator as illustrated in Figure 4. It first initializes all binding objects that are going to be fuzzed via the predefined statements. It also stores the names of variables that reference initialized binding objects and their types as context information. After this initialization process, our test case generator goes into the `while` loop where it starts fuzzing targeted binding objects through JavaScript statements dynamically generated.

Favocado randomly selects targeted binding objects. The number of binding objects should be decided heuristically but related objects must be selected together. Also, the duration of fuzzing the selected binding objects needs to be decided heuristically. In our evaluation, we managed the total number of objects, including related objects, for which Favocado generates test cases less than 6.

Generating test cases. When Favocado generates a JavaScript statement, it randomly selects a statement format from the list shown in Figure 5. Our statement formats contain basic forms to use binding objects (defining an object, assigning a value to a property, and calling a method) as well as forms to use prototype objects. Also, Favocado can generate iterative statements, arrays, and functions dynamically.

Once a format is selected, Favocado completes a selected statement format by using the semantic and context information. For example, when a format shown in Figure 5 is selected, Favocado randomly picks a binding object and checks whether the object is available through the `eval` API which will return an error if the object is not available. If the object is unavailable (because Favocado invoked a method that deallocates the object before making this statement), Favocado replaces the variable name of this object from the context information after initializing the object again. Favocado, thus, can prevent reference errors by not accessing objects which do not exist.

Next, Favocado randomly selects a method and its arguments. Basically, Favocado sets proper values for arguments, but, it randomly sets wrong types to find type confusion bugs in binding code. This principle also applies to when Favocado uses the other statement formats. To take another example, when creating a statement such as "`obj.prop = value`", Favocado intentionally can set a wrong type to the `value` so that `value` has a different type from the `prop`. In addition, Favocado tests the other types of bugs by intentionally setting wrong values (*e.g.*, values go beyond the possible range) and previously freed objects. It is worth noting that Favocado can prevent unexpected runtime errors by finding and assigning correct types but it randomly makes runtime errors on purpose for finding bugs such as type confusion bugs. Lastly, Favocado prepares a variable to store a return value from the selected method by creating a new variable.

Generating functions in test cases. When Favocado generates a function while fuzzing, it fills the function body by randomly picking and completing statement formats. A return statement of a function can return either nothing or a value. When a return statement returns a value, a type of the return value can be the same type of properties of targeted objects in the test case generator so that functions generated by Favocado can be used to set properties of targeted objects. For example, when Favocado generates a function, if the `HTMLDialogElement` object shown in Listing 2 is the only targeted object in the test case generator, Favocado inserts a return statement that returns either a Boolean type or a `DOMString` type object because properties of the targeted object have only the two possible types.

V. Evaluation

In this section, we first evaluate the latest semantics-aware JavaScript engine fuzzer, CodeAlchemist, to test how many semantically correct test cases using binding objects could be generated. We, then, evaluate the effectiveness of Favocado based on its ability to find distinct bugs in various types of binding code and JavaScript runtime systems.

Q1. Are existing JavaScript engine fuzzers sufficient to fuzz JavaScript binding code?
Q2. Can Favocado discover new vulnerabilities in real-world JavaScript runtime systems?
Q3. Can Favocado be applied to fuzzing different types of binding code in JavaScript runtime systems?
Q4. How does Favocado compare to state-of-the-art JavaScript fuzzers that can fuzz binding code?

A. Experimental Setup

Implementation. Favocado consists of two modules: (1) the IDL/API reference parser in Python and (2) the main fuzzing controller in Python and JavaScript. For implementing the parser, we modified a publicly available IDL parser.[1]. The parser parses and stores the semantic information of binding code in JavaScript data types. Also, we implemented an API reference parser for parsing the Acrobat API Reference in Python [7] for fuzzing PDF binding objects of the Adobe Acrobat Reader and Foxit Reader which are closed-source applications.

After extracting the semantic information of binding code, the main fuzzing controller analyzes relationships between binding objects. It, then, randomly picks binding objects based on the relationships and creates a test case generator, as a JavaScript file, that contains the semantic information of selected binding objects as well. The test case generator runs on a JavaScript runtime system, fuzzing targeted binding objects.

Targeted JavaScript Runtime Systems. To evaluate the effectiveness of Favocado, we selected 4 targets as follows:

T1. Adobe Acrobat Reader (PDF): It uses the AcroJS engine that contains binding code to support customizing of PDF files through JavaScript code.
T2. Foxit PDF Reader (PDF): It contains the V8 JavaScript engine to support PDF files that embed JavaScript code.
T3. Chromium (Mojo and DOM): An open-source browser using the V8 JavaScript engine. We performed fuzzing on Mojo and DOM binding objects of the Chromium browser with Favocado.
T4. WebKit (DOM): A browser engine used by the Safari browser. It uses the JavaScriptCore engine. We conducted fuzzing on DOM objects of the WebKit engine.

Our 4 targets are currently used by a very large number of users. In addition, they have been tested rigorously by security analysts because their vulnerabilities can directly be a threat to users. Therefore, we believe that previously unknown and distinct bugs found by Favocado can represent the effectiveness of Favocado's bug-finding ability.

Counting distinct bugs. The ultimate way to measure the performance of a fuzzer is showing the number of distinct bugs [32]. Because we evaluated Favocado with the most recent version of target systems, all bugs found by Favocado were previously unknown ones (there is no ground truth). Therefore, to prevent overcounting distinct bugs, we *manually* analyzed all crashes that occurred while fuzzing target programs. We counted a distinct bug only if an instruction pointer address (where a crash occurred) was different from the others and a unique series of minimized JavaScript statements caused a crash. We reported all distinct bugs to vendors with our

[1]https://chromium.googlesource.com/chromium/src/tools/idl_parser/+/884cd8ee4e59bc46d72f300de217215f81a5af72/idl_parser.py

		Breakdown of Runtime Errors		
Success Rate	**Fail Rate**	Syntax Error	Reference Error	Type Error
28.24%	71.76%	1.76%	34.80%	63.44%

TABLE I: Evaluation results of the modified CodeAlchemist as a PDF binding object fuzzer. Only 28.24% of test cases could run without causing a runtime error, resulting in 0 crashes.

analysis reports. Also, we verified that there were no instances of misclassification through our manual inspection of every PoC that we reported to vendors.

B. *Evaluation of CodeAlchemist*

Before evaluating Favocado, we performed experiments using a state-of-the-art JavaScript engine fuzzer, CodeAlchemist, in order to examine its suitability as a binding code fuzzer. CodeAlchemist is the latest *semantics-aware* generative JavaScript engine fuzzer that focuses on generating more semantically valid test cases using code blocks disassembled from a large corpus [27]. We, especially, evaluated how many semantically correct test cases using PDF binding objects can be generated by CodeAlchemist.

To this end, we first provided syntactically and semantically valid JavaScript code using PDF binding objects as seeds. (CodeAlchemist used regression tests from repositories of the four major JavaScript engines, and test code snippets from Test262 [1]) The PDFs were collected from Mozilla's PDF.js test suite [4] which contains over 600 PDF files embedding JavaScript code that uses various PDF binding objects for testing PDF rendering engines. Also, we collected PDFs from Virustotal [6] where we can find numerous malicious PDFs that use PDF binding objects. Because JavaScript test cases using PDF binding objects are not prevalent, we had to find code snippets from malicious PDFs for providing more seeds. In total, we collected 22,180 PDFs. We, then, refined the collected JavaScript samples by using the Esprima [2], which resulted in 4,450 valid JavaScript code snippets. In addition, we used 4,197 V8 regression test suite samples [5] as seeds. Consequently, we prepared 8,647 valid JavaScript snippets as seeds.

We set CodeAlchemist with the default parameters as in the repository. However, the dynamic analysis step to determine types will not be able to recognize PDF binding objects defined by PDF Readers. Therefore, to provide CodeAlchemist the best possible chance of utilizing the seeds that exercise PDF binding objects, we modified CodeAlchemist's dynamic analysis module to recognize these objects as a generic JavaScript object type. We accomplished this by implementing a proxy tarpit [48] for each PDF binding object. We generated 100K JavaScript test cases through CodeAlchemist configured for fuzzing PDF binding code and each test case was embedded into a pdf file.

We loaded the 100K PDF files, each of which embeds a test case generated by CodeAlchemist, on the Adobe Acrobat Reader v2019.012.20040. Experimental results are shown in Table I. Only 28.24% of test cases succeeded to execute without causing a runtime error but could not make a crash. About 71% of test cases could not execute completely mostly because of reference errors and type errors (these bugs are 98.24%)—because of CodeAlchemist's inability to generate semantically correct test cases.

C. *Favocado on PDF Binding Objects*

JavaScript in PDF files is widely used to customize documents. We selected two PDF viewers (Adobe Acrobat Reader v2019.012.20040 and Foxit Reader v9.5) that implement PDF binding objects embedded in a JavaScript engine. Both PDF readers are closed-source programs, and thus, we could not directly parse IDL files. Therefore, we parsed the Adobe API reference [7] for building the semantic information of PDF binding objects so that we can start fuzzing them. It is worth noting that there was no openly available API reference documents for the Foxit reader. We, thus, had to use the semantic information extracted from the Adobe API reference [7] for the Foxit reader.

Adobe Acrobat Reader. For fuzzing Adobe Acrobat Reader, we used 8 virtual machines (VMs)—2 cores and 4GB of memory for each VM (1 fuzzing process executes on each VM). We set Favocado to select less than 6 objects (not including the related objects) and generate test cases only for each group of selected objects. We ran the test case generator by setting it to change targeted objects if no crash has been found up to 100K JavaScript statements. This fuzzing campaign continued for 2 weeks.

In total, we found 39 distinct bugs within just 2 weeks. Among them, 18 bugs are exploitable ones that can cause a serious security problem such as an arbitrary code execution as shown in Table III. Also, 11 vulnerabilities have become CVE entries by the time of writing. The vendor acknowledged that the impact of those vulnerabilities is "critical."

Foxit Reader. We, also, performed fuzzing on PDF objects against another PDF viewer—Foxit reader—for 3 days. Except for the fuzzing duration, we employed the same setting as we used for fuzzing Adobe Acrobat Reader. Even though not all test cases generated by Favocado were semantically correct (because we used the semantic information extracted from the Adobe API reference for the Foxit reader), we found 3 distinct use-after-free vulnerabilities in the Foxit reader.

D. *Favocado on Mojo and DOM Objects of Chromium*

To evaluate the effectiveness of Favocado for other binding objects, we prepared and performed fuzzing on Mojo and DOM binding objects in Chromium v84.0.4110.0 with Favocado. We used the same environment as we used for fuzzing PDF binding objects except for the process of extracting the context information.

DOM binding objects. Because Chromium is an open-source browser, we parsed IDL files that defines interfaces of DOM and constructed the semantic information of them. We, then, let Favocado run for 2 weeks on 8 VMs. As a result of fuzzing, we found 6 distinct bugs including 2 security vulnerabilities.

Mojo binding objects. Mojo is a platform-agnostic library that enables Inter Process Communication (IPC) between processes implemented in multiple programming languages. Chromium implements the Mojo library and provides Mojo binding objects to JavaScript.

We first modified the Mojom Parser[2] to parse and construct the semantic information of Mojo binding objects that Favocado can process. Because the Chromium browser has implemented the dedicated Mojom parser, we did not use our IDL parser even though IDL files for Mojo are available. We, then, conducted fuzzing with Favocado for 1 week on 8 VMs. Consequently, Favocado discovered 2 distinct bugs including an exploitable use-after-free vulnerability.

E. Favocado on DOM Objects of WebKit

Domato fuzzer has discovered around 40 vulnerabilities from DOM binding objects of the WebKit within 2 years [21]. Even with this long-term fuzzing campaign, we performed fuzzing on DOM objects of the most recent version of the WebKit browser engine v2.28 as of when we conducted this evaluation, rather than evaluating Favocado against an old version of the WebKit. This fuzzing campaign lasted for 4 days on 8 VMs with the same evaluation setup used for fuzzing the other JavaScript runtime systems.

While fuzzing DOM objects of WebKit for 4 days, Favocado discovered 3 new distinct bugs. All of these bugs are exploitable security vulnerabilities.

F. Comparison with Domato

To demonstrate the effectiveness of Favocado, we compared it with Domato, a state-of-the-art binding code fuzzer using manually developed context-free grammars. To this end, we used Favocado and Domato for fuzzing the newest version of Adobe Acrobat Reader v2020.009.20067 as of when we started this evaluation. This version contained patches for fixing bugs that we had reported.

To evaluate Domato, we implemented a grammar file for PDF binding objects. Especially, we constructed the grammar only for the `Field` and `Doc` objects because all bugs that we found from Adobe Acrobat Reader v2019.012.20040 are related to these objects. We note that this setting based on our fuzzing results helps Domato to find bugs more effectively by reducing the input space a lot—by generating test cases only for those objects. We let Domato to run for 1 week on 8 VMs, fuzzing the binding objects. We, then, started another fuzzing campaign with Favocado for 1 week. For making a fair comparison, we only let Favocado to fuzzing for the `Field` and `Doc` binding objects. In addition, we did not count bugs found by Favocado if the bugs are the same one that we already discovered in the previous version of Adobe Acrobat Reader.

Domato discovered 1 use-after-free vulnerability, while Favocado discovered 6 distinct bugs including the one found by Domato. Furthermore, we check the error rate of test cases that Domato generated: 65.36% of test cases successfully executed (around 34% of test cases caused runtime errors). This comparison result demonstrates that Favocado outperforms the state-of-the-art binding code fuzzer in terms of finding distinctive vulnerabilities.

[2] The Mojom Parser. https://chromium.googlesource.com/chromium/src/mojo/+/refs/heads/master/public/tools/mojom

	Success Rate	**Fail Rate**	Breakdown of Runtime Errors		
			Syntax Error	Ref. Error	Type Error
Chromium	90.92%	9.08%	6.55%	18.97%	74.48%
WebKit	90.75%	9.25%	6.31%	21.81%	71.87%

TABLE II: Runtime errors that Favocado's test cases raised while fuzzing binding object of Chromium and WebKit.

```
1 x = {}
2
3 x.toString = function(){
4   this.flattenPages(0);
5   return "center";
6 }
7
8 textfield = this.addField("Field", "text", 0,
      [0,0,800,800]);
9 textfield.alignment = x
```

Listing 4: Minimized JavaScript snippet for triggering a use-after-free vulnerability (CVE-2019-8211) on Adobe Acrobat Reader.

G. Runtime errors of Favocado

Favocado dynamically generates JavaScript statements based on the semantic information extracted from binding code and the context information maintained while fuzzing to prevent unexpected runtime errors. However, when Favocado generates statements calling a method and assigning a value to a property of a method, Favocado randomly triggers runtime errors by intentionally using different types of objects, values out of range, *etc.* for finding vulnerabilities. We set Favocado to have 20% chance of generating such erroneous statements as default.

We measured how many runtime errors occurs while fuzzing binding objects on Chromium and WebKit. To this end, we created 100,000 JavaScript statements for each browser through Favocado and monitored execution results of them. Table II shows the experimental results. Overall, about 10% of JavaScript statements caused runtime errors and most of them are type errors.

H. Case Study

To illustrate how Favocado finds vulnerabilities in binding code, we introduce four real-world vulnerabilities discovered by Favocado. Given JavaScript snippets are minimized test cases that Favocado generated while fuzzing each target.

CVE-2019-8211. Listing 4 shows a minimized JavaScript snippet that can trigger a use-after-free bug on Adobe Acrobat Reader v2019.012.20035.

Favocado first defined an object `x`. It, then, assigned a function to the `toString` method of the `x` object. To this end, on Line 4–5, Favocado defined a function where `flattenPages` API is called with an argument `0`. This API call enforces all Field objects on page 0 of a PDF file will be deallocated. After calling the API, the custom `toString` function returns a string "`center`." Favocado knew that `toString` should return a string type. Moreover, Favocado made the return statement because the `alignment`

No.	Target JavaScript Runtime System	Type	Exploitable	Impact	Status
1	Adobe Acrobat Reader v2019.012.20040	Use-after-free	✓	High	CVE-2019-8211
2	Adobe Acrobat Reader	Use-after-free	✓	High	CVE-2019-8212
3	Adobe Acrobat Reader	Use-after-free	✓	High	CVE-2019-8213
4	Adobe Acrobat Reader	Use-after-free	✓	High	CVE-2019-8214
5	Adobe Acrobat Reader	Use-after-free	✓	High	CVE-2019-8215
6	Adobe Acrobat Reader	Use-after-free	✓	High	CVE-2019-8220
7	Adobe Acrobat Reader	Use-after-free	✓	High	CVE-2019-16448
8	Adobe Acrobat Reader	Use-after-free	✓	High	CVE-2020-3792
9	Adobe Acrobat Reader	Use-after-free	✓	High	Reported
10	Adobe Acrobat Reader	Untrusted pointer dereference	✓	High	CVE-2019-16446
11	Adobe Acrobat Reader	Heap out-of-bound write	✓	High	CVE-2020-9594
12	Adobe Acrobat Reader	Heap out-of-bound read	✓	Moderate	Reported
13	Adobe Acrobat Reader	Uninitialized heap memory use	✓	Moderate	Reported
14	Adobe Acrobat Reader	Uninitialized heap memory use	✓	Moderate	Reported
15	Adobe Acrobat Reader	Uninitialized heap memory use	✓	Moderate	Reported
16	Adobe Acrobat Reader	Type confusion	✓	High	CVE-2019-8221
17	Adobe Acrobat Reader	Type confusion	✓	High	*Fixed
18	Adobe Acrobat Reader	Type confusion	✓	High	*Fixed
19	Adobe Acrobat Reader	Null pointer dereference	✗	Low	Reported
...	Adobe Acrobat Reader	Null pointer dereference	✗	Low	Reported
39	Adobe Acrobat Reader	Null pointer dereference	✗	Low	Reported
40	Adobe Acrobat Reader v2020.009.20067	Use-after-free	✓	High	CVE-2020-9722
41	Adobe Acrobat Reader	Use-after-free	✓	High	Reported
42	Adobe Acrobat Reader	Heap overflow	✓	High	Reported
43	Adobe Acrobat Reader	Heap out-of-bout read	✓	Moderate	Reported
44	Adobe Acrobat Reader	Uninitialized heap memory use	✓	Moderate	Reported
45	Adobe Acrobat Reader	Null pointer dereference	✓	Moderate	Reported
46	Foxit Reader v9.5	Use-after-free	✓	High	Reported
47	Foxit Reader	Use-after-free	✓	High	Reported
48	Foxit Reader	Use-after-free	✓	High	Reported
49	Chromium (Mojo) v84.0.4110.0	Use-after-free	✓	High	Reported
50	Chromium (Mojo)	Null pointer dereference	✗	Low	Reported
51	Chromium (DOM) v84.0.4110.0	Heap overflow	✓	High	CVE-2020-6524
52	Chromium (DOM)	Security check fail	✓	Moderate	Reported
53	Chromium (DOM)	Null pointer dereference	✗	Low	Reported
...	Chromium (DOM)	Null pointer dereference	✗	Low	Reported
56	Chromium (DOM)	Null pointer dereference	✗	Low	Reported
57	WebKit v2.28	Use-after-free	✓	High	Reported
58	WebKit	Heap out-of-bound Write	✓	High	Reported
59	WebKit	Heap out-of-bound Read	✓	Moderate	Reported
60	WebKit	Null pointer dereference	✗	Low	Reported
61	WebKit	Null pointer dereference	✗	Low	Reported

***Fixed** = The vendor silently fixed a bug after we reported it.

TABLE III: Distinct bugs found by Favocado. We have performed fuzzing on 4 different JavaScript runtime systems. As a result, Favocado found 61 distinct bugs. Among 61 distinct bugs, there are 33 exploitable vulnerabilities. The high impact implies that it can lead attackers to have arbitrary code execution on a target system. Bugs that have the moderate impact can be used with other bugs for information leakage or arbitrary code execution. The low impact bugs can cause a crash but cannot be exploited for other attacks other than the denial-of-service attack.

property of the `Field` object must have one of specific string values (*i.e.*, left, right, or center).

After allocating a new text field object on Line 8, the JavaScript engine called the `toString` method of the `x` object, which was customized by Favocado, to cast the type of `x` to a string type object because the alignment property is a string type. At this moment, the text field object had been deallocated by calling the `flattenPages` API. Therefore, when the JavaScript engine assigned the return value of the customized `toString` function, a use-after-free error occurred. This case clearly demonstrates the effectiveness of semantically correct JavaScript statements that Favocado generates for finding bugs.

Type confusion vulnerability in PDF binding objects. The JavaScript snippet shown in Listing 5 triggers a crash using

```
1 myColor = this.getColorConvertAction();
2 myColor.convertProfile = this.bookmarkRoot;
3
4 this.bookmarkRoot.remove();
5 this.colorConvertPage(0, [myColor], []);
```

Listing 5: Minimized JavaScript snippet for triggering a type confusion vulnerability on Adobe Acrobat Reader.

a type confusion vulnerability on Adobe Acrobat Reader v2019.012.20035. On Line 1, `getColorConvertAction` API returns a `colorConvertAction` object. Next, Favocado generated a JavaScript statement that assigns the `bookmarkRoot` property to the `convertProfile` property of the `colorConvertAction` object. Types of the two properties are different: The `bookmarkRoot` property is the

```
1 smsRcv_A = new blink.mojom.SmsReceiverPtr();
2 Mojo.bindInterface(blink.mojom.SmsReceiver.name,
      mojo.makeRequest(smsRcv_A).handle);
3
4 smsRcv_B = new blink.mojom.SmsReceiverPtr();
5 Mojo.bindInterface(blink.mojom.SmsReceiver.name,
      mojo.makeRequest(smsRcv_B).handle);
6
7 smsRcv_A.receive()
```

Listing 6: Minimized JavaScript snippet for triggering a use-after-free vulnerability on Chromium.

```
1 f = this.addField("test", "button", 0,
      [200,0,0,-50]);
2 f.borderStyle = border.d;
3 f.borderColor = color.blue;
4 this.zoom = 3179;
5 f.borderWidth = 2147483647;
6 this.pageNum = 0;
7 this.scroll(25);
```

Listing 7: Minimized JavaScript snippet for triggering a heap out-of-bound write vulnerability (CVE-2020-9594) on Adobe Acrobat Reader.

`Bookmark` object type and the `convertProfile` is a string type. Therefore, the assignment should not have been allowed but executed without causing a runtime error. Consequently, calling the `remove` method of the `bookmarkRoot` on Line 4 resulted in deallocation of the `colorConvertAction` object created on Line 1. And, on Line 5, calling the `colorCovertPage` API caused a crash because the second argument—the `myCololr` variable—was deallocated.

Even though a crash occurred on Line 5 (the implementation of the `colorCovertPage` API did not check the liveness of arguments), the root cause of this vulnerability is the assignment of a wrong type on Line 2. Favocado intentionally generated the statement to find a type confusion bug but the JavaScript engine did not raise a runtime error, as we discussed in Section IV-B. As this case shows, it is necessary and important to not only generate semantically correct test cases but also examine possible bugs through generating erroneous statements when fuzzing binding code.

Use-after-free vulnerability in Mojo binding objects. Listing 6 is a JavaScript snippet that triggers a use-after-free vulnerability in the mojo binding objects of Chromium v84.0.4110.0. On Line 1–2, Favocado allocated a `SmsReceiverPtr` object and bound it as an interface in order to receive data. On Line 3–4, Favocado allocated a new `SmsReceiverPtr` object (`smsRcv_B`) and replaced it with the previous interface (`smsRcv_A`) by calling the same API (`Mojo.bindInterface`), which resulted in deallocation of the previous interface in the binding code (in the native code). However, in the execution context that the JavaScript engine manages, the object was still alive and accessible. Consequently, accessing the deallocated object on Line 7 could raise a use-after-free bug.

This case illustrates that Favocado is able to find vulnerabilities that misrepresentations of binding objects between the JavaScript engine and the binding code can cause. Even though the object had been deallocated by the binding code, in the perspective of Favocado, the Line 7 was a semantically correct JavaScript statement. If the object (`smsRcv_A`) was not accessible, Favocado would never have called the `receive` API because it always checks whether or not every object used in a statement is accessible in JavaScript side (except for when it generates erroneous statements).

CVE-2020-9594. Favocado found a heap out-of-bound write vulnerability in the Adobe Acrobat Reader v2019.012.20035. The minimized JavaScript statements generated by Favocado are shown in Listing 7. The JavaScript code snippet simply assigns specific values to the properties of the button field object after allocating it on Line 1. In the following lines, Favocado simply assigns randomly generated values to various properties of the button field object. Favocado, then, calls the `scroll` API with an integer argument on Line 7.

Because the Adobe Acrobat Reader is a closed-source program, we could not find the exact root cause of this vulnerability within our limited analysis time but the vendor acknowledged that this vulnerability can cause arbitrary code execution. This case shows the importance of chaining together a multitude of semantically correct statements that creates and manipulates an object to find JavaScript binding bugs.

I. *Summary*

In summary, as shown in Table III, we found 61 new and distinct bugs from 4 different JavaScript runtime systems within short fuzzing campaigns (from just 3 days to 2 weeks for each target). This result clearly demonstrates our approach, Favocado, can play an important role to find bugs in binding code of various JavaScript runtime systems.

VI. Discussion

In this section, we discuss the limitations of Favocado and future work.

Implementation detail. There are several limitations in the current implementation of Favocado to enable fully automated fuzzing on binding code of JavaScript engines.

First off, even though we implemented and used a parser that can extract the semantic information of binding code from API references (*e.g.*, Acrobat API Reference [7]), we could not completely avoid manual effort to construct the semantic information of binding code that Favocado can process. Specifically, we added missing data and wrong values that our parser failed to parse. Also, we need to manually implement the JavaScript statements for initializing some binding objects especially for some binding objects that require environment-specific data. although a lot of binding objects can be initialized directly. We leave the complete semantic information construction from API references and automated binding object initialization as future work.

Feedback-driven fuzzing. One disadvantage of Favocado is the redundancy of test cases, which stems from the random mutation strategy. Albeit Favocado showed the noticeable effectiveness with a random mutation strategy, we believe that we can improve it by adopting feedback-driven approaches such as the coverage-guided fuzzing for generating test cases. DIE, Nautilus, and Superion showed that mutational approaches combined with code coverage feedback for generating highly structured inputs can increase the effectiveness of fuzzing [8, 40, 57]. However, incorporating such feed-back driven approaches for fuzzing binding code, while generating semantically correct test cases, is an open research question that we need to explore.

Minimizing test cases for analyzing crashes. The test case generator of Favocado itself is a JavaScript program that runs on a JavaScript runtime system, fuzzing binding code. Unlike existing JavaScript engine and binding code fuzzers, which deliver a series of JavaScript statements and evaluate it repeatedly, a fuzzing session started by Favocado continues until it finds a crash or stops the fuzzing for changing a test case generator. Therefore, the total amount of JavaScript statements that need to be analyzed for finding minimized test cases is substantial. In this work, we used on an open-source JavaScript test case reducer such as Lithium [3] to find minimized test cases, which is fairly effective but a time-consuming manual analysis for each crash is inevitable. We need an automated approach for analyzing test cases so that the entire fuzzing process can be more efficient.

Fuzzing binding code in other scripting languages. In this paper, we proposed Favocado as a JavaScript binding code fuzzer and demonstrated its effectiveness through various evaluations. Even though we focused on JavaScript binding code, we believe that our approach can be applied to fuzzing binding code of the other scripting languages such as Perl and Python. Favocado does not rely on a specific feature of JavaScript, it rather is designed for fulfilling the requirements for fuzzing binding code in general as we discussed in Section III-A. Therefore, extending our approach to the other scripting languages would be straightforward.

VII. Related work

Fuzzing has been one of the most active research areas in the security field. For enhancing the effectiveness of fuzzing, each fuzzer capitalizes different strategies. Rebert *et al.* [44] proposed seed selection methods and demonstrated how we can choose the best seeds by mathematically reasoning them. A lot of fuzzers obtains control flow graphs, states of program, and coverage information through dynamic tracing or static analysis and leverage them to identify flow-dependent data fields or to generate inputs that can explore new execution paths [9, 10, 13, 14, 17, 20, 23, 30, 38, 42, 49, 59]. Also, security researchers have been leveraging symbolic execution, concolic execution, and taint tracking methods to solve complex constraints so that fuzzers can explore deep execution paths in programs and identify specific parts of inputs that affect execution paths [16, 18, 25, 28, 43, 51, 60, 61]. Moreover, there have been many proposed approaches to tackle challenges in fuzzing operating systems such as the non-determinism and complex data structures by leveraging specific hardware components, hypervisors or static/dynamic analysis methods [15, 19, 26, 31, 39, 46, 50, 52, 58]. On the other hand, Neuzz [47] utilized neural network models to predict branching behaviors of programs, and then, it predicts critical locations of program inputs so that it can perform more mutations on such critical locations to achieve high edge coverage.

A common limitation of those fuzzers is the lack of ability to generate syntactically and semantically correct input. Therefore, many fuzzers proposed to overcome the limitation and to enable effective fuzzing programs which require highly structured input such as JavaScript. Generating well-formed (syntactically valid) input is the first requirement for fuzzing such programs. Domato [21] and jsfunfuzz [45] generate test cases by using manually developed context-free rules. Superion [57] and Nautilus [8] leveraged code coverage feedback and context-free grammars together to improve the effectiveness of fuzzing. Superion and Nautilus showed that the coverage-guided input generation approach using context-free grammars allows us to explore deep and wide execution paths with highly structured input. However, these fuzzers based on manually developed grammars are lacking in the capacity to generate semantically correct cases. In contrast to them, Favocado is designed to generate syntactically and semantically correct test cases to enable very effective fuzzing on binding code without using manually developed grammars.

There are JavaScript engine fuzzers that focused on utilizing existing test suites and PoC exploits of previous vulnerabilities to enable. Montage [34] is a JavaScript engine fuzzer guided by a neural network language model (NNLM). It transforms abstract syntax trees (ASTs) of existing JavaScript test cases into a sequence of AST subtrees so that they can be trained by a NNLM. DIE [40] proposed two mutation strategies: structure-preserving mutation and type-preserving mutation. They collected JavaScript files including previous vulnerabilities' exploits and used them as seeds. Favocado prioritizes constructing new combinations of APIs with various arguments and new access patterns to targeted binding objects, rather than leveraging known vulnerabilities and existing test suites.

To enable semantics-aware fuzzing, Skyfire [56] was proposed to generate well-structured test cases through the probabilistic context-sensitive grammar that specifies syntax features and semantic rules learned from existing samples. CodeAlchemist [27] also focused on generating semantics-aware test cases through code blocks disassembled from a large corpus to correctly use variables based on their types. Albeit they showed better performance in resolving semantic errors than the other fuzzers, they could not achieve generating semantically correct test cases over semantics-aware ones. Meanwhile, Favocado can continuously perform fuzzing targeted binding objects through semantically correct test cases, which led us to discover a lot of high-impact security vulnerabilities.

VIII. Conclusion

In this paper, we propose Favocado, a JavaScript binding code fuzzer, that can generate semantically correct test cases. Favocado first extracts the semantic information of binding objects by parsing IDL files or API references. Then, Favocado selects binding objects based on their relationships for fuzzing them only on a single fuzzing session. Once fuzzing started, Favocado generates and executes test cases based on the extracted semantic information as well as the context information that Favocado dynamically manages. Through our evaluation, we demonstrated the importance of semantically correct test cases and the effectiveness of Favocado: within not a long fuzzing campaign, we have discovered 61 previously unknown security vulnerabilities in 4 different JavaScript runtime systems (3 different types of binding code).

Acknowledgment

We would like to express our gratitude to the anonymous reviewers for their thoughtful reviews and our shepherd, Cornelius Aschermann.

This material is based upon work supported in part by the National Science Foundation (NSF) under grant 1651661 and 2000792, the Defense Advanced Research Projects Agency (DARPA) under agreement number HR001118C0060 and FA875019C0003, the Army Research Office and accomplished under MURI grant number W911NF-17-1-0370, the Office of Naval Research (ONR) KK1847, the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2017-0-00168, Automatic Deep Malware Analysis Technology for Cyber Threat Intelligence), and a grant from the Center for Cybersecurity and Digital Forensics (CDF) at Arizona State University.

Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of United States Government or any agency thereof.

References

[1] *ECMAScript Language test262*, https://v8.github.io/test262/website/default.html.

[2] *Esprima: ECMAScript parsing infrastructure for multipurpose analysis*, https://esprima.org.

[3] *Lithium: An automated testcase reduction tool*, https://github.com/MozillaSecurity/lithium.

[4] *PDF.js: A general-purpose, web standards-based platform for parsing and rendering PDFs*, https://mozilla.github.io/pdf.js/.

[5] *V8 Testing*, https://v8.dev/docs/test.

[6] *VirusTotal*, https://www.virustotal.com/.

[7] *JavaScript for Acrobat API Reference*, May 2015, https://www.adobe.com/content/dam/acom/en/devnet/acrobat/pdfs/AcrobatDC_js_api_reference.pdf.

[8] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for Deep Bugs with Grammars," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[9] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with Input-to-State Correspondence," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[10] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[11] 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 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[12] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[13] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[14] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: towards a desired directed grey-box fuzzer," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[15] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[16] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[17] P. Chen, J. Liu, and H. Chen, "Matryoshka: fuzzing deeply nested branches," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[18] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[19] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[20] J. De Ruiter and E. Poll, "Protocol State Fuzzing of TLS Implementations," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, D.C., Aug. 2015.

[21] I. Fratric, *Domato: A DOM fuzzer*, (accessed Mar 19, 2020), https://github.com/googleprojectzero/domato.

[22] M. Furr and J. S. Foster, "Checking type safety of foreign function calls," in *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun. 2005.

[23] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[24] 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 2nd International Conference on Informatics & Applications (ICIA)*, Lodz, Poland, Sep. 2013.

[25] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," in *Proceedings of the 22nd USENIX Security Symposium (Security)*, Washington, D.C., Aug. 2013.

[26] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[27] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[28] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to Fuzz from Symbolic Execution with Application to Smart Contracts," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

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

[30] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang, "Instrim: Lightweight instrumentation for coverage-guided fuzzing," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[31] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

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

[33] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. S. McKinley, "Jinn: synthesizing dynamic bug detectors for foreign language

interfaces," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Canada, Jun. 2010.

[34] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[35] S. Li and G. Tan, "Finding bugs in exceptional situations of JNI programs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, Nov. 2009.

[36] ——, "Finding reference-counting errors in Python/C programs with affine analysis," in *Proceedings of the European Conference on Object-Oriented Programming 2014 (ECOOP'14)*, Uppsala, Sweden, Jul.–Aug. 2014.

[37] MozillaSecurity, *funfuzz*, (accessed Mar 2, 2020), https://github.com/MozillaSecurity/funfuzz.

[38] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[39] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing {OS} Fuzzer Seed Selection with Trace Distillation," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[40] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing JavaScript Engines with Aspect-preserving Mutation," in *Proceedings of the 41th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

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

[42] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: fuzzing by program transformation," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[43] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

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

[45] J. Ruderman, *Releasing jsfunfuzz and DOMFuzz*, (accessed Mar 2, 2020), http://www.squarefree.com/2015/07/28/releasing-jsfunfuzz-and-domfuzz.

[46] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug. 2017.

[47] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[48] P. Snyder, C. Taylor, and C. Kanich, "Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[49] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[50] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[51] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[52] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: Facilitating dynamic analysis of device drivers of mobile systems," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[53] G. Tan and J. Croft, "An Empirical Security Study of the Native Code in the JDK," in *Proceedings of the 17th USENIX Security Symposium (Security)*, San Jose, CA, Jul.–Aug. 2008.

[54] G. Tan and G. Morrisett, "ILEA: Inter-language analysis across Java and C," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA)*, Montreal, Canada, Oct. 2007.

[55] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *Proceedings of the 21st European Symposium on Research in Computer Security(ESORICS)*, Heraklion, Greece, Sep. 2016.

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

[57] ——, "Superion: Grammar-aware greybox fuzzing," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Montréal, Canada, May 2019, pp. 724–735.

[58] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[59] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[60] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[61] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.