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 Doup´e*, and Yan Shoshitaishvili*
*Arizona State University, +North Carolina State University, ^PayPal, Inc., -Samsung Research

NDSS 2021

Minkyung Park

mkpark@mmlab.snu.ac.kr

March 16, 2022

Contents

- JavaScript binding code
- Fuzzing challenges
- Favocado design
- Evaluation
- Conclusion

JavaScript and its fuzzing

- JavaScript (JS) is a dynamic language interpreted by JS engines
  - e.g., Chrome V8, SpiderMonkey, Chakra, etc.
- The use of JS has expanded into the entire computing ecosystem
  - Adobe Acrobat utilizes JS engines to provide dynamic or interactive content through JS code embedded in PDF documents
- It is difficult to effectively fuzz JS engines because of the language's **syntactical** correctness
  - JS engines parse user input code into an abstract syntax tree (AST) and then process the tree
  - User inputs that cannot be transformed into an AST are easily rejected before being processed
- Existing fuzzers use context-free grammars or existing correct test cases

JavaScript and its fuzzing

- It is also important to generate semantically correct JS codes

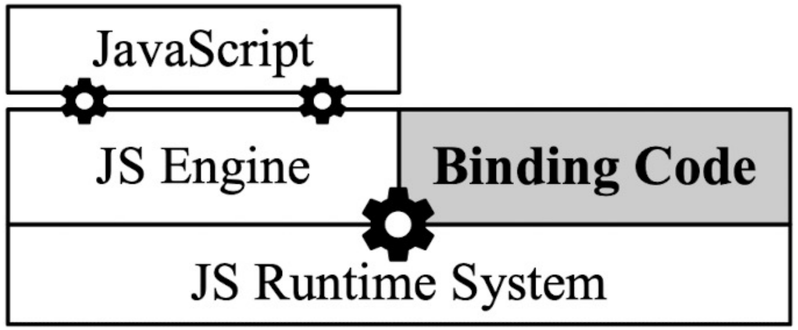
- Many JS statements have interdependent relationships
  - Correct use of method names, argument types, and return types
  - e.g., not using after free

- Most JS fuzzers cannot generate fully semantically correct test cases
  - Some fuzzers generates semantic-aware test cases, but the percentage of rejected test cases is a significant problem
    - E.g., CodeAlchemist

- The problem becomes more severe in *JS binding layers*

JavaScript binding layer

- JS engines provide a binding layer to provide functionality implemented in unsafe languages such as C and C++
- JS cannot be used to directly implement low-level functionalities and those are implemented in native code
  - e.g., memory management and file access
- JS binding code translates data representation
  - It creates and maps data types between JS and native code
  - Then, JS scripts can call native functions or control data of native components
  - e.g., DOM object
  - During the translation, type- and memory-safety features cannot be implemented



Challenges to fuzz JS binding layer (1)

- It is practically impossible to generate legitimate JS test cases
- Fuzzers need to input many JavaScript statements as a basic testing unit
  → A semantically incorrect test case has to stop executing and retire
- Typical JS test cases that trigger the execution of binding code *once* raise the excessive number of JavaScript exceptions
  - It involves two steps (i) creating the object and (ii) setting a property or calling a function
- To fuzz the binding layers, a fuzzer should generate ***syntactically and semantically correct*** test cases to eliminate runtime exceptions

Challenges to fuzz JS binding layer (2)

- The input space is enormous
- There are many object types that are accessible through the binding layer as a DOM
  - Each DOM object may have a multitude of methods and properties
- Creating all objects to enumerate all properties and manipulate all methods is simply infeasible
- An effective fuzzer should be able to optimize the test case generation by ***reducing the size of the input space***

Favocado approach

- Favocado is a new fuzzing approach to find vulnerabilities in the binding layers of JS engines
- Generating legitimate test cases
  - Favocado parses semantic information from the Interface Definition Language (IDL) files or API references to obtain semantic information
    - Such as exact types and possible values of binding objects
  - Favocado uses the information to complete a fuzzed statement and prevents unexpected runtime errors
- Reducing input spaces

Favocado approach

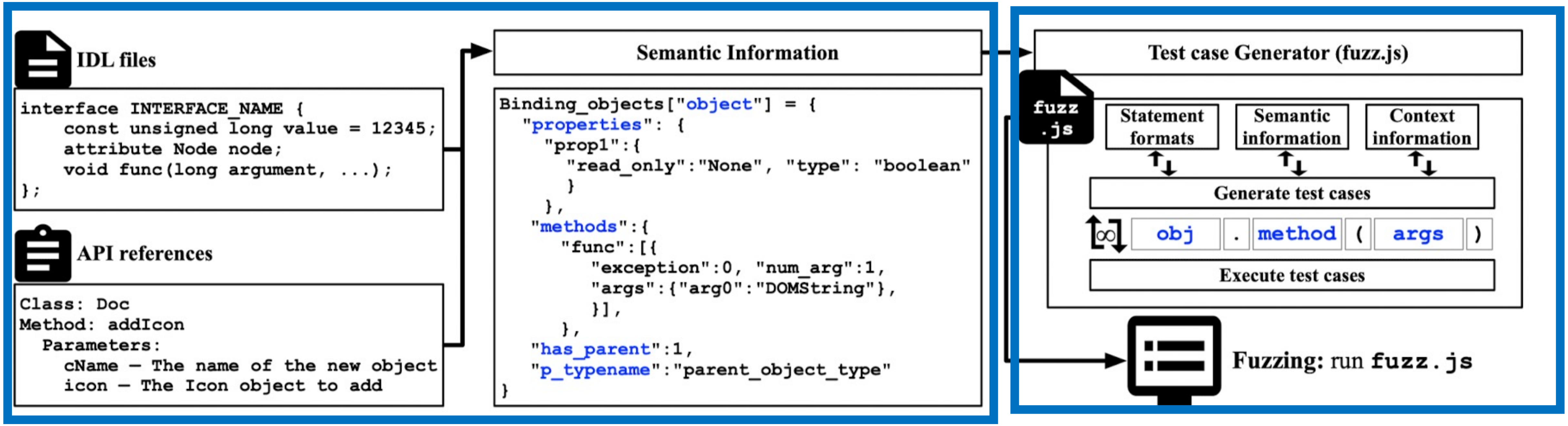
- Favocado is a new fuzzing approach to find vulnerabilities in the binding layers of JS engines

- Generating legitimate test cases
- Reducing input spaces
  - One unique feature of the JavaScript binding layer is the relative isolation of different DOM objects
  - Different DOM objects are implemented as separate native modules unless an object in a module can be used by code in another module
    - E.g., spell.check() in Adobe Acrobat’s spell module and Net.HTTP.request() in its Net.HTTP module

```
spell.check()           =   Net.HTTP.request()
Net.HTTP.request()          spell.check()
```

  - Based on the relations between objects, the entire input space is divided into equivalence classes
  - Favocado only mutate within each equivalence class

Favocado overview



Semantic information construction

Dynamic test case generator

Semantic information construction

- Favocado extracts
- (1) Binding object names
  - A name of each object and a name of a parent
- (2) Binding object methods
  - Each method’s name and all arguments’ types
  - Checks whether a method can raise an exception
- (3) Binding object properties
  - A name, type, and possible string values of each property
  - 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"
}
```

Semantic information construction

- Favocado finds binding objects related each other using semantic information

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

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

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

- By the relation between objects, the entire input spaces can be divided into equivalence classes

Test case generator

- Test case generator (fuzz.js) dynamically generates and executes JS statements inside a target JS engine
- It includes the semantic information, context information, statement formats, and pre-defined JS statements,

- Context information: a list of *allocated* variable names with their types
  - The generator maintains the context information to prevent unexpected runtime errors
  - E.g., reference and type errors

- Statement formats

Statement formats

```
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 }
```

Test case generator

- Test case generator (fuzz.js) dynamically generates and executes JS statements inside a target JS engine
- It includes the semantic information, context information, statement formats, and pre-defined JS statements,

- Pre-defined JS statements
  - To manually initialize some binding objects that cannot be initialized automatically
  - Usually, binding objects that require environment-specific data such as IP address or image files

Test case generator

fuzz.js

```
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
   }
}
```

- For setup, Favocado randomly selects a set of targeted binding objects
  - The related objects also should be selected
- Firstly, it initializes all objects that are going to be fuzzed via predefined statements
- It randomly selects a statement format
- Then, it completes the format using the semantic information and the context information

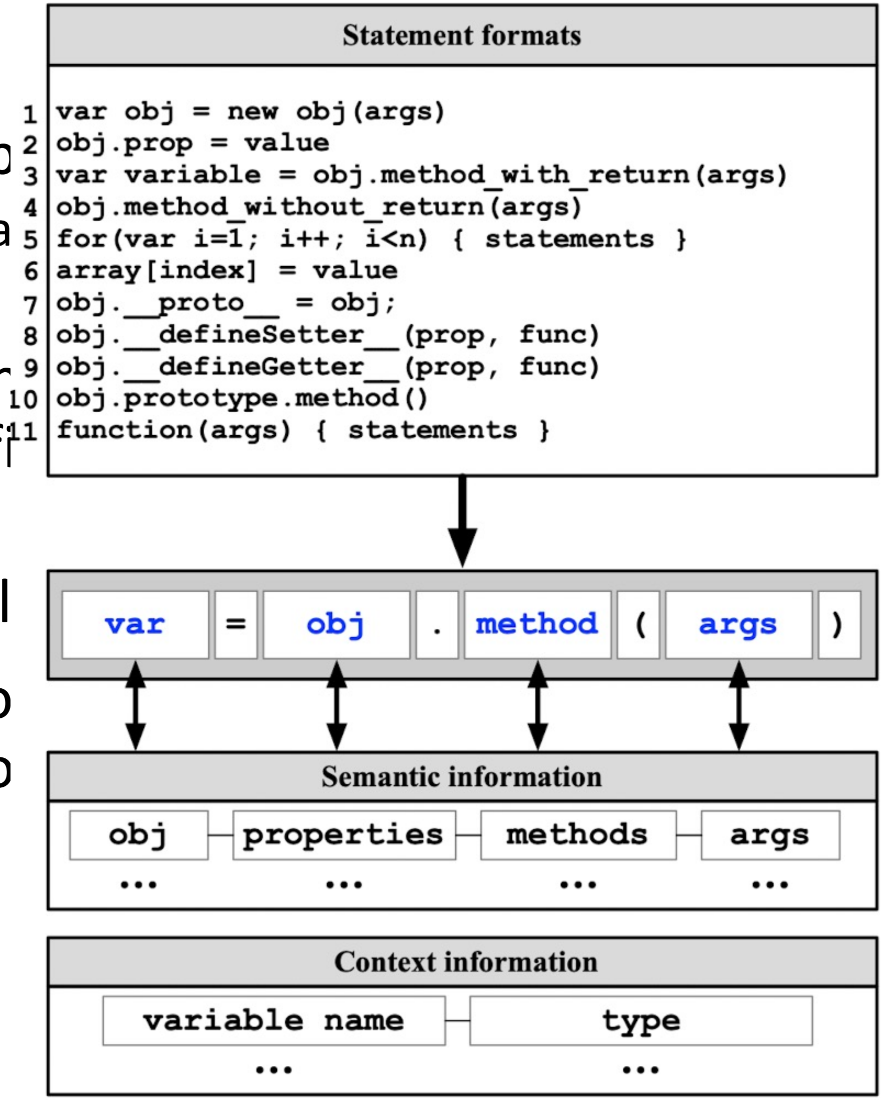
Test case generator

```
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
    }
}
```

fuzz.js

- For setup, … f targeted binding ob…
  - The rela…
- Firstly, it ir… to be fuzzed via predefi…
- It randoml…
- Then, it co… antic informatio…



Test case generator

fuzz.js

```
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
    }
}
```

- For setup, Favocado randomly selects a set of targeted binding objects
  - The related objects also should be selected
- Firstly, it initializes all objects that are going to be fuzzed via predefined statements
- It randomly selects a statement format
- Then, it completes the format using the semantic information and the context information

Evaluation

- **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?

Experiment setup

- Implementation
  - An IDL parser based on a chromium parser and API parsers for parsing PDF readers

- System and parameter setup
  - 8 VMs: 2 cores and 4GB of memory for each VM
  - Set to select less than 6 object

- Targeted JS runtimes (recent versions are used)
  - PDF readers: Adobe Acrobat Reader and Foxit PDF Reader
  - Chromium (Mojo and DOM) and WebKit (DOM)

- Counting distinct bugs: to prevent overcounting, the authors manually analyzed all crashes
  - Counted 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

Suitability of Favocado

- CodeAlchemist is a state-of-the-art JavaScript engine fuzzer that focuses on generating valid test cases
- How many semantically correct test cases can be generated shows the suitability as a binding code fuzzer
- Among 100K test cases, 28% were valid without causing a runtime error but could not make a crash
  - From 8,647 seed files, 100K test cases were generated

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

CodeAlchemist

			Breakdown of Runtime Errors		
	Success Rate	**Fail Rate**	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%

Favocado

Distinct bugs found by Favocado

- Adobe Acrobat Reader
  - 39 bugs within just 2 weeks
- Foxit Reader
  - 3 use-after-free vulnerabilities
- Chromium
  - For DOM binding objects, 6 bugs including 2 vulnerabilities within 2 weeks
  - For Mojo binding objects, 2 bugs including one vulnerability within 1 week
- WebKit
  - 3 bugs for 4 days

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.

Case study: Chromium

- Mojo is a platform-agnostic library that enables Inter Process Communication (IPC) between processes implemented in multiple programming languages

```
smsRcv_A = new blink.mojom.SmsReceiverPtr();
Mojo.bindInterface(blink.mojom.SmsReceiver.name,
     mojo.makeRequest(smsRcv_A).handle);

smsRcv_B = new blink.mojom.SmsReceiverPtr();
Mojo.bindInterface(blink.mojom.SmsReceiver.name,
     mojo.makeRequest(smsRcv_B).handle);

smsRcv_A.receive()
```

Resulted in deallocation of the smsRcv_A

Minimized JavaScript snippet for triggering a use- after-free vulnerability on Chromium

Conclusion

- The paper proposes Favocado, a novel fuzzer for JavaScript binding code
- It can generate semantically correct test cases by using semantic information extracted from IDL files or API references
- It also dynamically handles runtime exceptions using the context information
- The evaluation shows its effectiveness by finding 61 vulnerabilities in 4 different JS runtimes