Fuzzing JavaScript Environment APIs with Interdependent Function Calls

Renáta Hodován, Dániel Vince, and Ákos Kiss(✉)

Department of Software Engineering, University of Szeged,
Dugonics tér 13., Szeged 6720, Hungary
{hodovan,vinced,akiss}@inf.u-szeged.hu

Abstract. The prevalence of the JavaScript programming language makes the correctness and security of its execution environments highly important. The most exposed and vulnerable parts of these environments are the APIs published to the executed untrusted JavaScript programs. This paper revisits the fuzzing technique that generates JavaScript environment API calls using random walks on so-called prototype graphs to uncover potentially security-related failures. We show the limits of generating independent call expressions, the approach of prior work, and give an extension to enable the generation of interdependent API calls that re-use each other's results. We demonstrate with an experiment that this enhancement allows our approach to exercise JavaScript environment APIs in ways that were not possible with the previous approach, and that it can also trigger more issues in a real target.

1 Introduction

> *"For the seventh year in a row, JavaScript is the most commonly used programming language"*
>
> —Stack Overflow Developer Survey Results 2019

The popularity of JavaScript (standardized as ECMAScript [4]) has been steadily growing over the past few years until it became the most commonly used programming language. Not only is it enabling the client-side execution of web applications in browsers but it is also driving the server-side [21]. Moreover, even resource-constrained IoT devices have become programmable using JavaScript, thanks to memory-aware execution engines [14,26] and application environments built on top of them [23].

Having an easy to learn, well-supported, and flexible language available on all our computing devices opens up great possibilities but it also makes the correctness and security of its execution environments highly important. This is especially true when untrusted code can be executed in these environments, e.g., in web browsers, on modular platforms with a community-maintained package

W. Ahrendt and S. L. Tapia Tarifa (Eds.): IFM 2019, LNCS 11918, pp. 212–226, 2019.
https://doi.org/10.1007/978-3-030-34968-4_12

registry, or on devices with JavaScript-based open application models [5]. In these scenarios, those parts of the execution environments are the most exposed and the most vulnerable, which publish such APIs to the executed JavaScript programs that cross the so-called trust boundary, i.e., which allow calling and passing parameter data from untrusted JavaScript programs into the trusted code of the environment that is typically compiled to machine code and running at elevated privileges.

Random or fuzz testing is a popular automatic technique for uncovering bugs with potential security implications [25]. Fuzzers generate totally or partially random test cases and feed them to their target (a.k.a. system-under-test or SUT) hoping that some of these inputs cause the SUT to malfunction, e.g., lead to an exploitable crash. In the context of JavaScript execution environments (engines, platforms) this means that fuzzers have to generate executable JavaScript programs as test inputs.

In previous work [9], we introduced a graph representation modelling the objects and types of JavaScript engine APIs, and a fuzzing approach for generating test cases consisting of call expressions to those APIs by performing random walks on such graphs. In this paper, we show the limits of generating independent call expressions and extend the previous work to enable the generation of interdependent API calls that re-use each other's results. We demonstrate that this extension can better exercise APIs in a JavaScript environment and trigger more issues in a real project than the original variant.

The rest of the paper is organized as follows: first, in Sect. 2, we give a brief overview of the used graph representation and we also give an algorithm for generating (independent) call expressions from the graph, then in Sect. 3 we describe how to allow the calls to be interdependent. In Sect. 4, we present the results of our experiment with the proposed approach. In Sect. 5, we discuss related work. Finally, in Sect. 6, we give a summary of our work and conclude the paper.

2 Prototype Graphs

In our previous paper [9], we introduced a graph representation for the weak type system of JavaScript, called the *Prototype Graph*. The graph balances between the theoretical possibility that each object in a JavaScript program can have completely different prototype and members, and the observation that in practice they tend to fall into similarity categories in an actual execution environment. As the rest of this paper builds on this representation, for the sake of completeness, we give the original definition of prototype graphs below.

Definition 1 (Prototype Graph). Let a *Prototype Graph* be a labeled directed multigraph (a graph allowing parallel edges with own identity)

$$G = \langle V, E, s, t, l_{prop}, l_{param} \rangle$$

such that

- $V = V_{type} \cup V_{sig}$, set of vertices, where the subsets are disjoint,
  - V_{type} vertices represent 'types', i.e., categories of similar objects,
  - V_{sig} vertices represent 'signatures' of callable types, i.e., functions,
- $E = E_{proto} \cup E_{prop} \cup E_{cstr} \cup E_{call} \cup E_{param} \cup E_{ret}$, set of edges, where all subsets are mutually disjoint,
  - E_{proto} edges represent prototype relation ('inheritance') between types,
  - E_{prop} edges represent the properties ('members') of types,
  - E_{cstr} and E_{call} edges connect callable types to their signatures and represent the two ways they can be invoked, i.e., the construct and call semantics,
  - E_{param} edges represent type information on parameters of callable types,
  - E_{ret} edges represent return types of callable types,
- $s : E \to V$ assigns to each edge its source vertex, under the constraint that $\forall e \in E_{proto} \cup E_{prop} \cup E_{cstr} \cup E_{call} \cup E_{param} : s(e) \in V_{type}$ and $\forall e \in E_{ret} : s(e) \in V_{sig}$,
- $t : E \to V$ assigns to each edge its target vertex, under the constraint that $\forall e \in E_{proto} \cup E_{prop} \cup E_{ret} : t(e) \in V_{type}$ and $\forall e \in E_{cstr} \cup E_{call} \cup E_{param} : t(e) \in V_{sig}$,
- the $\langle V, E_{proto}, s|_{E_{proto}}, t|_{E_{proto}} \rangle$ directed sub-multigraph is acyclic,
- $l_{prop} : E_{prop} \to \Sigma$ labeling function assigns arbitrary symbols ('names') to property edges, under the constraint that $\forall e_1, e_2 \in E_{prop} : s(e_1) = s(e_2) \Rightarrow l_{prop}(e_1) = l_{prop}(e_2) \iff e_1 = e_2$,
- $l_{param} : E_{param} \to \mathbb{N}_0$ labeling function assigns numeric indices to parameter edges, under the constraint that $\forall e_1, e_2 \in E_{param} : t(e_1) = t(e_2) \Rightarrow l_{param}(e_1) = l_{param}(e_2) \iff e_1 = e_2$.

Informally, a prototype graph is a collection of *type* and *sig* vertices connected by six different kind of edges (and multiple edges can run between two vertices). *Proto* and *prop* edges connect *type* vertices, while the others connect *type* and *sig* vertices in one direction or the other. And finally, member (property) name information and function argument order is encoded in edge labels. Vertices have no labeling as all relevant information is encoded in the existence of and labels of edges.

How such a graph can be built automatically is described both in our previous paper and will also be discussed later in Sect. 4. But Fig. 1 already shows an example prototype graph of 7 *type* and 4 *sig* vertices, manually constructed based on a portion of the ECMAScript 5.1 standard [4, Sections 15.2, 15.3]. The graph contains the types of `Object`, `Object.prototype`, `Function`, and `Function.prototype` objects, the global object, two constructor signatures for `Object`, and also the types and call signatures for two additional functions (`Object.create` and `Object.valueOf`).

In our previous work, we have shown how this graph representation encodes property accesses, type-correct parametrization of function and constructor calls, and also how the expressive power of the graph can be extended to deal with literals; and we have formally defined a set of function call expressions that can be generated from a graph. Now, we rephrase our original formal definitions into an algorithm to show how to generate a single function call instead

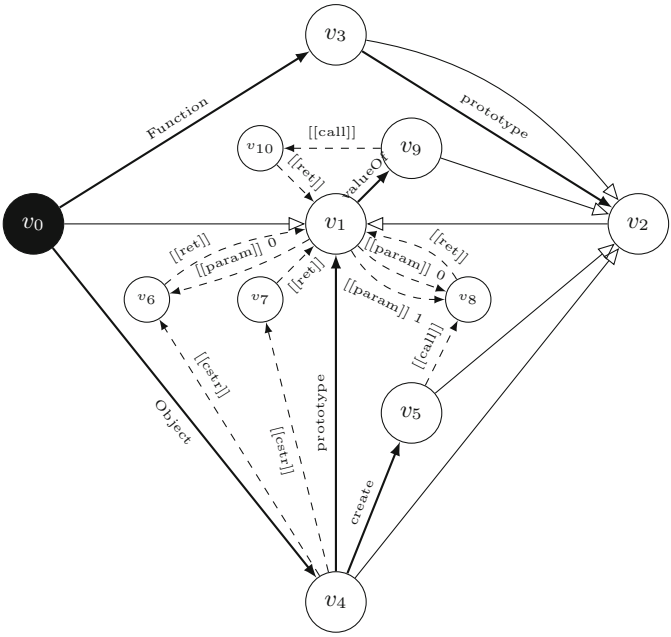


Fig. 1. Example prototype graph manually constructed based on a portion of the ECMAScript 5.1 standard. Large and small nodes represent *type* and *sig* vertices respectively. The single black node on the left represents the type of the global object. (Both colors and vertex labels are for identification and presentation purposes only.) Thick lines with labels represent *prop* edges, thin lines with hollow arrows represent *proto* edges, while dashed lines with double-bracketed labels represent *cstr*, *call*, *param*, and *ret* edges.

of all possible functions function call expressions. The random and recursive graph walking algorithm that collects the labels of visited edges thus generating a test case is named RANDOMCALL and is given in Listing 1. The informal concept behind the formal definition is to first walk forward on *proto*, *prop*, *cstr*, *call*, and *ret* edges to a *sig* vertex, then walk backward on *param* and *proto* edges, and so on... For the sake of brevity, G is always an alias to $\langle V_{type} \cup V_{sig}, E_{proto} \cup E_{prop} \cup E_{cstr} \cup E_{call} \cup E_{param} \cup E_{ret}, \quad s, t, l_{prop}, l_{param} \rangle$ in the algorithm, as in Definition 1; Λ is a function from *type* vertices to a set of literals (i.e., valid JavaScript expressions lying outside the expressiveness of the graph); and v_0 is a designated starting vertex in V_{type}, usually the type of the global object of the JavaScript language. The algorithm is also relying on some helper functions: RANDOMPATH finds a random finite path in a graph between two vertices on *proto*, *prop*, *cstr*, *call*, and *ret* edges and returns the list of edges on the found path; PARAMETERS enumerates the sources of all incoming *param* edges in ascending order of the edge labels (and returns them together with the edge labels); DESCENDANTS returns all vertices transitively available backwards

Listing 1. Algorithm for generating random function call expressions from a prototype graph.

```
1  procedure randomExpr(G, Λ, v_from, v_to)
2    expr := random(Λ(v_from))
3    forall e_step in randomPath(G, v_from, v_to) do
4      if e_step ∈ E_prop then
5        expr += '.' + l_prop(e_step)
6      elif e_step ∈ E_call ∪ E_cstr then
7        if e_step ∈ E_cstr then
8          expr := 'new (' + expr + ')'
9        end if
10       expr += '('
11       forall n, v_param in parameters(G, t(e_step)) do
12         v_param := random(descendants(G, v_param))
13         if |Λ(v_param)| > 0 and random({true,false}) then
14           expr += random(Λ(v_param))
15         else
16           expr += randomExpr(G, Λ, v_from, v_param)
17         end if
18         expr += not last iteration ? ',' : ''
19       end forall
20       expr += ')'
21     end if
22   end forall
23   return expr
24 end procedure

26 procedure randomCall(G, Λ, v_0)
27   return randomExpr(G, Λ, v_0, random(V_sig))
28 end procedure
```

from a starting point (including the starting vertex); and random randomly selects one element from its parameter set. Finally, + and += stand for string concatenation.

As an example, below we give some test cases that can be generated with randomCall(G_{ex}, Λ_{ex}, v_0), where G_{ex} is the graph shown in Fig. 1, v_0 is the type of the global object in that graph (i.e., the `this` of the current lexical scope, marked with black), and $\Lambda_{ex} = \{v_0 \mapsto \{\texttt{'this'}\}, v_1 \mapsto \{\texttt{'\{}\texttt{'}\}\}$:

- `this.Function.valueOf()`,
- `new (this.Object)(this.Function.valueOf())`,
- `this.Object.valueOf().create(this.Object.prototype.valueOf(),{})`.

Listing 2. Datagram sending example.

```
var client = require('dgram').createSocket('udp4');
client.send(Buffer.from('Some bytes'), 41234,
    'localhost', function (err) { client.close(); });
```

3 Interdependent Function Calls

The above described prototype graph representation and its use to generate test cases for JavaScript engines showed promising results as they had triggered real failures [9]. However, the previous paper has only shown the results of generating expressions that are independent of each other. As it is also shown by the examples at the end of the previous section, even if such expressions were executed in sequence in the same execution context, there was very little possibility for them to have an effect on each other (a notable exception is if an expression changes the properties of a prototype object, as that may have an overarching effect even on future descendants of the prototype).

The fact that the expressions are mostly independent is no shortcoming if the types or objects of the API-under-test can be exercised that way, i.e., without a state carried across multiple expressions. This turns out to be mostly the case for the standard built-in ECMAScript objects [4, Section 15], i.e., if a JavaScript engine is only tested in its purest form (like the *jsc*, *d8*, *jerry*, or *duk* command line utilities of the WebKit/JavaScriptCore, V8, JerryScript, or Duktape projects, respectively). However, JavaScript engines are rarely used on their own, they are usually embedded in some bigger application. To make the embedding useful, the environments or platforms that build on top of JavaScript engines extend the standard ECMAScript API with custom types and objects and functions; and we have found that independent function call expressions are insufficient for the testing of many of these APIs.

For example, both Node.js [21] and IoT.js [23] are JavaScript application environments that allow the extension of the execution context with various modules. Both environments support UDP datagrams via the *dgram* module (as the resource-constrained IoT.js aims at being upward compatible with the desktop and server-targeted Node.js platform), where the proper sending of a datagram requires four steps: the loading of the module, the creation of a socket, the sending of the message, and the closing of the socket. As the example in Listing 2 shows, the established API of the *dgram* module does not allow to express all these steps as a single expression. In this example, the result of the expression that created the socket object needs to be carried over to and reused in the next expression: the same object must be used to send the message as well as to close the socket. However, if expressions are generated independently, there is no way for an object that was created in one expression to be further accessed in another. This means that the RANDOMCALL algorithm has no chance to generate test cases like Listing 2.

Listing 3. Algorithm for generating lists of interdependent random function call expressions from a prototype graph.

```
1  randomExpr′ := randomExpr
2  procedure randomExpr(G, Λ, v_from, v_to)
3    return randomExpr′(G, Λ, random(domain(Λ)), v_to)
4  end procedure

6  procedure randomCallList(G, Λ, n)
7    list := ''
8    forall i in 1..n do
9      uid := uniqueID()
10     v_from := random(domain(Λ))
11     v_to := random(V_sig)
12     list += 'var ' + uid + '=' + randomExpr′(G, Λ, v_from, v_to) + ';'
13     Λ(ret(G, v_to)) ∪= {uid}
14   end forall
15   return list
16 end procedure
```

The *dgram* module is not the only one with an API that needs objects kept across functions calls, and Node.js or IoT.js are not the only platforms that host such APIs. The classic embedders of JavaScript engines, i.e., browsers, and their web API also show similar patterns. Rendering context objects of the DOM interface of HTML canvas elements [29] are also typical examples of objects that have to be reused in multiple function calls.

Therefore, we propose to enhance the prototype graph-based fuzzing approach shown in the previous section by generating multiple expressions for a single test case, capturing the results of the individual expressions, and re-using them in following generations. Fortunately, the prototype graph-based formalism and algorithm have an 'extension point', the Λ function, that allows the generation of expressions that lay outside the expressiveness of the graph. So, we propose to generate variable statements with the graph-generated expressions being the initialisers (and with unique identifiers as variable names), and to change (update) the Λ function after the generation of every variable statement so that it extends the set, which is associated with the type of the variable, with the identifier of the variable. This way the Λ function (and the fuzzing technique) becomes capable of generating not only literals as starting points and parameters of call expressions but also variable references, thus opening the possibility for interdependent API function calls that reuse the result of each other.

The above outlined idea is formalized by the randomCallList algorithm in Listing 3. The algorithm is using some further helper functions in addition to those already introduced: uniqueID returns a valid unique JavaScript identifier on every call, and ret returns the target vertex of an outgoing *ret* edge. Additionally, domain returns the set for which its parameter function is defined,

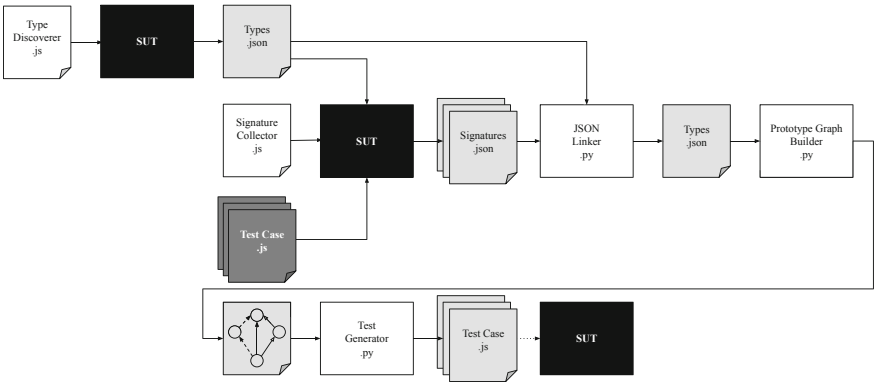


Fig. 2. Architecture overview of the prototype implementation of the JavaScript API fuzzing approach. The white elements are part of the implementation, while the black boxes stand for the SUT. The dark and light gray elements are inputs used and outputs generated during their execution.

while the $\Lambda(.) \cup=$ notation stands for the update of the Λ function, extending its result set (as well as its domain, potentially). (The redefinition or wrapping of RANDOMEXPR at lines 1–4 ensures that the recursion at line 16 of Listing 1, which generates parameters for function calls, can also make use of the updated Λ function.)

To follow up on the example of the previous section, the following code snippets are test cases that can be generated with RANDOMCALLLIST on G_{ex} and Λ_{ex} (with various n inputs):

```
– var v0=this.Object.valueOf();
  var v1=this.Object.prototype.valueOf();
  var v2=v0.create(v1,{});
– var v0=this.Object.valueOf();
  var v1=v0.create(v0,v0);
```

4 Experimental Results

To experiment with prototype graph-based fuzzing and with the approaches that generate independent and interdependent function calls, we have created a prototype implementation that is able to build prototype graphs by automatically discovering the API of its SUT and to generate test cases from the built graph using the algorithms shown above. The architecture overview of the prototype implementation is shown in Fig. 2.

The top part of the architecture overview outlines the automatic graph building steps. To get information about the basic structure of the API of the SUT, the implementation makes use of the introspecting capabilities of the JavaScript language. It uses a carefully crafted engine-agnostic JavaScript program, which – when executed once by the target environment – looks at the global object of the SUT (the `this` of the current lexical scope), retrieves its prototype object (using `Object.getPrototypeOf`) as well as its properties (using `Object.getOwnPropertyNames`), and does so to all found objects transitively, thus *discovering* the target's API. Moreover, the program can also distinguish between regular and callable objects (i.e., functions) and can record the length of the argument list of the visited functions (as given by their `length` property).

The so-retrieved information is still vague about the signatures of the discovered API functions because it finds only the number of arguments without any type details. Therefore, the prototype implementation contains another specifically crafted script that can be loaded into the target environment before the execution of other programs, and wraps all API functions discovered in the previous step to record the types of parameters and return values of actual invocations. With sufficiently many and diverse executions, the result of this *signature collection* step can be used to gather information about possible valid parametrizations of the API.

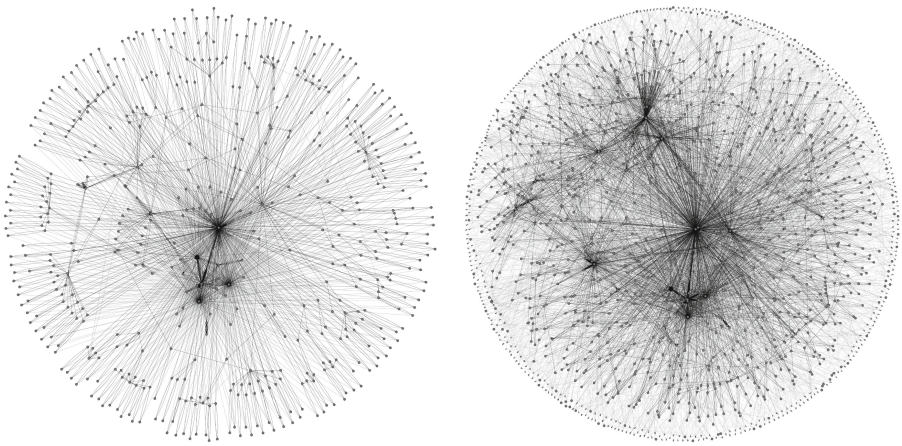
As the final phase in the graph building process, the implementation links together the information provided by the structure discovery and signature collection steps and builds the data structure that conforms to Definition 1. (Fig. 2 also discloses the implementation detail that those parts of the prototype tool that are executed in the SUT – i.e., discovery and signature collection – are written in JavaScript, while the rest of the steps – like linking the results of the previous steps and building the actual graph, as well as the test case generation from the graph – are written in Python.)

As environment-under-test, we have chosen the IoT.js project [23], a modular application platform for embedded IoT devices built on top of the JerryScript [14] execution engine. We have executed both the discovery script in the environment and signature collection, using the project's own test suite to ensure that the discovered API functions are called diversely enough. Table 1 informs about the size of the so-built graphs (the numbers are given after both the discovery and signature collection steps), while Fig. 3 visualizes them. (Due to their size, the graph plots are not as clean as the manually created one in Fig. 1, but the graph resulting from the discovery step still shows some of the structure of the modular system of the IoT.js API. The graph extended with all the collected signature information is admittedly more entangled.)

The graph built as above (the one extended with all collected signature information) was used to generate test cases for IoT.js (outlined in the lower part of Fig. 2). Each generated test case contained either 20 independent API call expressions or 20 interdependent ones. The fuzzing session ran for 5 days and was driven by the Fuzzinator framework [11], which channelled the generated test cases to the SUT, monitored the execution, and collected unique failures

Table 1. Size metrics of prototype graphs built for IoT.js.

	After discovery step	Extended with collected signatures
$\|V_{type}\|$	576	1045
$\|V_{sig}\|$	8	425
$\|E_{proto}\|$	575	1044
$\|E_{prop}\|$	1569	2706
$\|E_{call}\|$	267	626
$\|E_{cstr}\|$	46	58
$\|E_{param}\|$	28	692
$\|E_{ret}\|$	8	425



(a) (b)

Fig. 3. Prototype graphs built for the JavaScript API exposed by IoT.js (a) using discovery only and (b) after extending discovery results with signatures seen in executed test cases.

(i.e., determined if multiple test cases triggered the same issue and kept only one of them).

By the end of the fuzzing session, the two test case generation variants have induced 14 different failures altogether in IoT.js. The test cases that contained independent function calls triggered 9 of the issues, while the test cases with interdependent function calls could trigger all of them. Manual investigation has revealed that two failures not found by the original approach could have been potentially triggered by test cases with independent function calls; it is due to the random nature of fuzz testing that they were not hit during our experiment. However, the manual investigation has also revealed that the other three test cases have actually made use of the interdependency of the generated function calls and thus could not have been generated by the original algorithm variant.

Table 2. Issues found in IoT.js by prototype graph-based fuzzing technique generating independent and interdependent function calls.

Issue ID	Independent calls	Interdependent calls
#1904	(✗)	✓
#1905	✓	✓
#1906	✓	✓
#1907	(✗)	✓
#1908	✗	✓
#1909	✓	✓
#1910	✗	✓
#1911	✗	✓
#1912	✓	✓
#1913	✓	✓
#1914	✓	✓
#1915	✓	✓
#1916	✓	✓
#1917	✓	✓

Listing 4. Issue #1908 of IoT.js.

```
var net = require('net')
var v0 = new (net.connect(1).constructor)()
try { v0.connect() } catch ($) { }
try { v0._handle.readStart() } catch ($) { }
```

Listing 5. Issue #1910 of IoT.js.

```
var http_common = require('http_common')
var v0 = http_common.createHTTPParser(1)
v0.execute(Buffer(6083374109688862375))
v0.resume()
```

The found failure-inducing test cases were reduced by the automatic test case reducer tool Picireny [10,12,15] and further beautified by hand, and the so-minimized inputs were reported to the issue tracker of IoT.js. Table 2 sums up these results, showing the public issue IDs of the found problems, and whether a test case generation technique has found it or not. (Check marks show issues found, crosses signal issues not found, while crosses in parentheses mark failures that could have been found, potentially.)

Listing 6. Issue #1911 of IoT.js.

```
1 var dgram = require('dgram')
2 var v0 = dgram.createSocket('udp4')
3 v0.addMembership(decodeURIComponent(), v0)
```

To highlight the results, we also show the test cases of those three issues that were found only by the algorithm that generated interdependent function calls (see Listings 4, 5, and 6). These examples demonstrate that the technique presented in this paper was able to exercise three separate parts of the API of its target environment in a way that was not possible with a previous approach.

5 Related Work

With the growing influence of the JavaScript language, its execution engines are also getting more attention security-wise.

Godefroid et al. published a grammar-based white-box methodology to test the JavaScript engine of Internet Explorer 7 [6]. They exploited the information gathered from symbolic execution of existing test cases and a context-free grammar definition of the input format, and created new test cases to exercise different control paths.

Ruderman created jsfunfuzz [20], a JavaScript fuzzer that manually defined generator rules to create syntactically and semantically correct test cases. This approach also took advantage of the introspection possibility of the language to extract field and method names but it did not try to build a complex model nor to infer function parametrization.

Holler et al. presented LangFuzz [13], a language-independent mutational approach that consists of two steps. First, it parses existing test cases and builds a so-called fragment pool from the created parse trees. After this preprocess step, LangFuzz creates new test cases from the extracted fragments by random recombinations. Although the idea is language-independent, the authors applied it to test SpiderMonkey, the JavaScript engine of the Firefox web browser where they found hundreds of bugs. IFuzzer [27] is an evolutionary approach built upon the idea of LangFuzz. It defines a fitness function using information about the generated test (like complexity metrics) and the feedback of the SUT (like crashes, exceptions, timeouts, etc.) to choose the elements of the next population. Similarly to LangFuzz, IFuzzer was applied to generate JavaScript sources and it exercised the SpiderMonkey engine. BlendFuzz [30] is also a similar solution to LangFuzz as it processes existing sources with ANTLR grammars, which also improves the mutation phase by collecting additional information from parse trees. SkyFire [28] is a tool aiming to generate valuable input seeds for other fuzzing strategies. It infers probabilistic context-sensitive grammars from examples to specify both syntax features and semantic rules and it uses this infor-

mation for seed generation. SkyFire was used to test the JavaScript engine of Internet Explorer 11.

Guiding random test generation by the SUT's coverage information has gained popularity recently. American Fuzzy Lop [31] (or AFL for short) is one of the most well-known example. It was used, adapted, and improved by many [1–3,16–18,22,24]. AFL is a language-independent mutation-based fuzzing approach. It usually starts with an initial test population and iteratively applies various atomic mutation operators (bit/byte insertion/deletion/replacement, etc.) on its elements. If a mutant covers new edges in the SUT then it will be kept for further iterations, otherwise it will be removed from the population. The main strength of this approach is that it is fast and it can be applied to any file-based SUT without being acquainted with the input format requirements. However, this can be a drawback, too, if the SUT has complex syntax requirements, like JavaScript engines, since it will generate many useless, syntactically incorrect test cases.

Honggfuzz [7] and libFuzzer [19] are also coverage-guided fuzzers that improve the fuzzing performance by running it in-process and by focusing only on some selected methods. Google's OSS-Fuzz [8] platform uses both AFL and libFuzzer to test open-source projects with a large user base.

6 Summary

In this paper, we have revisited a fuzzing (or random testing) technique that used prototype graphs to model the weak type system of the JavaScript programming language and generated function call expressions from such graphs to exercise the APIs of JavaScript execution engines. We have observed that JavaScript-based execution environments (built on top of execution engines) often exposed APIs that had parts which required multiple separate but not independent function calls to be reachable. As the original technique could not generate test cases for those parts of APIs, we have given an extension to the original graph-based approach so that it can generate a series of API calls which can re-use the results of each other.

We have created a prototype tool based on the here-presented algorithms and used it to fuzz test the IoT.js platform. The generated test cases have caused numerous issues in the platform-under-test: we have found 9 unique failures that were triggered by both algorithm variants (showing that the original approach is still a valid fuzzing technique), but there were also 5 unique failures that were caused by test cases generated by the new approach only. Manual investigation showed that at least 3 of these failure-inducing test cases indeed made use of interdependencies between multiple function calls and could not be merged into a single expression. All failures have been reported to the public issue tracker of the tested project along with the issue-triggering test cases.

Since the prototype implementation has already found real issues in a real project, we plan to further experiment with the technique. As a natural continuation of the current work, we plan to target the Node.js platform with this technique, as well as the JavaScript APIs of current web browsers. We foresee that these new targets may impose new requirements both on the implementation and on the formalism of the prototype graph (e.g., because of language constructs introduced by newer JavaScript specification versions they support). We also plan to adapt techniques that can guide prototype graph-based fuzzing – i.e., influence its randomness – as greybox techniques have been proved beneficial in other fuzzing approaches as well. Finally, although JavaScript is one of the most widespread programming languages these days, we would like to investigate the adaptability of the (proto)type graph-based API fuzzing technique to other languages as well.

Acknowledgment. This research was supported by the EU-supported Hungarian national grant GINOP-2.3.2-15-2016-00037 and by grant TUDFO/47138-1/2019-ITM of the Ministry for Innovation and Technology, Hungary.

References

1. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017), pp. 2329–2344. ACM (2017)
2. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as Markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016), pp. 1032–1043. ACM (2016)
3. Chen, Y., Jiang, Y., Liang, J., Wang, M., Jiao, X.: Enfuzz: From ensemble learning to ensemble fuzzing. Computing Research Repository abs/1807.00182 (2018)
4. Ecma International: ECMAScript Language Specification (ECMA-262), 5.1 edn., June 2011
5. Fitbit Inc: Fitbit OS. https://dev.fitbit.com. Accessed May 07 2019
6. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008), pp. 206–215. ACM (2008)
7. Google: honggfuzz. http://honggfuzz.com. Accessed 07 May 2019
8. Google: OSS-Fuzz - continuous fuzzing of open source software. https://github.com/google/oss-fuzz. Accessed 07 May 2019
9. Hodován, R., Kiss, Á.: Fuzzing JavaScript engine APIs. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 425–438. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_27
10. Hodován, R., Kiss, Á.: Modernizing hierarchical delta debugging. In: Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST 2016), pp. 31–37. ACM, November 2016
11. Hodován, R., Kiss, Á.: Fuzzinator: an open-source modular random testing framework. In: Proceedings of the 11th IEEE International Conference on Software Testing, Verification and Validation (ICST 2018), pp. 416–421. IEEE Computer Society, April 2018. https://github.com/renatahodovan/fuzzinator

12. Hodován, R., Kiss, Á., Gyimóthy, T.: Tree preprocessing and test outcome caching for efficient hierarchical delta debugging. In: Proceedings of the 12th IEEE/ACM International Workshop on Automation of Software Testing (AST 2017), pp. 23–29. IEEE, Buenos Aires, May 2017
13. Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: 21st USENIX Security Symposium, pp. 445–458. USENIX (2012)
14. JS Foundation, et al.: JerryScript. http://www.jerryscript.net. Accessed 07 May 2019
15. Kiss, Á., Hodován, R., Gyimóthy, T.: HDDr: a recursive variant of the hierarchical delta debugging algorithm. In: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST 2018). pp. 16–22. ACM, Lake Buena Vista, November 2018
16. Lemieux, C., Padhye, R., Sen, K., Song, D.: Perffuzz: automatically generating pathological inputs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, pp. 254–265. ACM, New York (2018)
17. Lemieux, C., Sen, K.: Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. Computing Research Repository abs/1709.07101 (2017)
18. Li, Y., Chen, B., Chandramohan, M., Lin, S.W., Liu, Y., Tiu, A.: Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017), pp. 627–637. ACM (2017)
19. LLVM Project: libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed 07 May 2019
20. Mozilla Security: jsfunfuzz. https://github.com/MozillaSecurity/funfuzz/. Accessed 07 May 2019
21. Node.js Foundation: Node.js. https://nodejs.org. Accessed 07 May 2019
22. Pham, V., Böhme, M., Santosa, A.E., Caciulescu, A.R., Roychoudhury, A.: Smart greybox fuzzing. Computing Research Repository abs/1811.09447 (2018)
23. Samsung Electronics Co., Ltd., et al.: IoT.js. http://www.iotjs.net. Accessed 07 May 2019
24. Stephens, N., et al.: Driller: augmenting fuzzing through selective symbolic execution. In: Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS 2016). Internet Society (2016)
25. Takanen, A., DeMott, J., Miller, C., Kettunen, A.: Fuzzing for Software Security Testing and Quality Assurance, 2nd edn. Artech House (2018)
26. Vaarala, S., et al.: Duktape. http://duktape.org. Accessed 07 May 2019
27. Veggalam, S., Rawat, S., Haller, I., Bos, H.: IFuzzer: an evolutionary interpreter fuzzer using genetic programming. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9878, pp. 581–601. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45744-4_29
28. Wang, J., Chen, B., Wei, L., Liu, Y.: Skyfire: data-driven seed generation for fuzzing. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 579–594. IEEE Press (2017). https://doi.org/10.1109/SP.2017.23
29. WHATWG: HTML living standard. https://html.spec.whatwg.org. Accessed 07 May 2019
30. Yang, D., Zhang, Y., Liu, Q.: Blendfuzz: a model-based framework for fuzztesting programs with grammatical inputs. In: IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 1070–1076. IEEE, IEEE Computer Society (2012)
31. Zalewski, M.: American fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed 07 May 2019