Fuzzing JavaScript Engine APIs

Renáta Hodován(✉) and Ákos Kiss

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

Abstract. JavaScript is one of the most wide-spread programming languages: it drives the web applications in browsers, it runs on server side, and it gets to the embedded world as well. Because of its prevalence, ensuring the correctness of its execution engines is highly important. One of the hardest parts to test in an execution environment is the API exposed by the engine. Thus, we focus on fuzz testing of JavaScript engine APIs in this paper. We formally define a graph representation that is suited to describe type information in an engine, explain how to build such graphs, and describe how to use them for API fuzz testing. Our experimental evaluation of the techniques on a real-life in-use JavaScript engine shows that the introduced approach gives better coverage than available existing fuzzing techniques and could also find valid issues in the tested system.

1 Introduction

JavaScript (standardized as ECMAScript [3] but rarely referred to by that name) is the de-facto standard programming language of web browsers, which have evolved into one of the most important application platforms of our days – in which evolution the language played a major role. However, it is not only the client side of the web where JavaScript spreads: it is gaining popularity in server-side scripting as well, thanks to the Node.js framework[1]. And recently JavaScript has penetrated the embedded world as well: several engines (e.g., Duktape[2], JerryScript[3]) and frameworks (e.g., IoT.js[4]) emerged that enable the programming of highly resource-constrained Internet-of-Things devices in JavaScript. Because of the prevalence of the language, ensuring the correctness of its execution engines – both functionally and security-wise – is of paramount importance.

One testing method that has a special focus on security is fuzzing [11] – or fuzz testing, random testing. In fuzzing, a so-called test generator produces totally random or partially randomized ('fuzzed') test inputs in a great volume, which

[1] https://nodejs.org/.
[2] http://duktape.org/.
[3] http://www.jerryscript.net/.
[4] http://www.iotjs.net/.

E. Ábrahám and M. Huisman (Eds.): IFM 2016, LNCS 9681, pp. 425–438, 2016.
DOI: 10.1007/978-3-319-33693-0_27

are then given to a program (or system-under-test, SUT) for processing in the hope that some of them cause malfunction. The most easily recognizable errors are crashes, assertion failures, and unhandled exceptions, since they certainly signal a design flaw in the application – quite often flaws that can be exploited by a malicious attacker. Since fuzzing has the potential of uncovering such errors, the technique is often used in internal security testing processes [8].

For a random test generation approach to be useful in practice, it should be able to reach as 'deep' in the SUT as possible, i.e., generate test inputs that are not discarded by early correctness (e.g., syntax or CRC) checks. For some input formats or SUTs, even the simplest, random byte sequence generation or existing test mutation (e.g., bit flipping) techniques may give satisfying results, but as the format gets stricter such approaches tend to scratch the surface of the SUT only. This is the case with JavaScript, too: even fuzzers with knowledge about the language syntax (e.g., AST mutators or grammar-based generators) are mostly exercising the parser of the engine-under-test only (i.e., whether language constructs are correctly recognized and transformed into internal representation). The reason for this is that the execution of these fuzzed inputs often fails because of mismatch between the generated operations and operands.

This is especially true for the application programming interfaces (APIs) exposed by the engines: there are requirements regarding what method can be invoked on what object with which arguments. Gathering this information manually from the standards is both burdensome and error prone. Moreover, both because of different stages in the implementation of (different versions of) the standard and because of different application domains (e.g., web browsers, server side, command line), the APIs exposed by existing engines differ (and will change as they evolve). Thus, automatic means for modeling the exposed API are heavily needed.

Existing approaches [1,10] that try to infer information – e.g., a type system – about JavaScript, however, work on user source code. Since the execution engine itself is rarely written in JavaScript, these methods cannot be used for engine analysis. To deal with the issues outlined above, we present three major novel contributions in this paper: first, we define a graph-based type representation that is suited to describe the API of a JavaScript execution engine with different precisions, then we describe two methods how to build such a graph, and finally, we show an application of the graph representation, i.e., how it can be used in fuzzing.

The rest of the paper is organized as follows: Sect. 2 gives the formal definition of the graph used throughout the paper. Section 3 formalizes API fuzzing based on the introduced graph representation. Section 4 outlines two automated methods to infer information about the API of an engine. Section 5 presents experimental results from a prototype implementation of the graph representation, the analyses, and the fuzzing technique. Section 6 discusses related work, and finally, Sect. 7 concludes the paper and forecasts future work.

2 Graph Representation of Type Information in JavaScript

It is a commonly known feature of JavaScript that although it has the concept of objects, it lacks a strong type system. Some kind of inheritance exists, mimiced by prototypes, but it is a relation between individual objects however. And theoretically, each object can be significantly different in terms of its prototype and members. Nevertheless, objects in practice in an actual execution environment tend to fall into similarity categories or 'types', i.e., they share the same prototype chain and they have members with the same name, which in turn again have the same 'type'. Below, we define a graph, titled the *Prototype Graph* after the prototype feature of the language, that can represent such type information.

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 $\forall e_1, e_2 \in E_{param} : t(e_1) = t(e_2) \Rightarrow l_{param}(e_1) = l_{param}(e_2) \iff e_1 = e_2$.

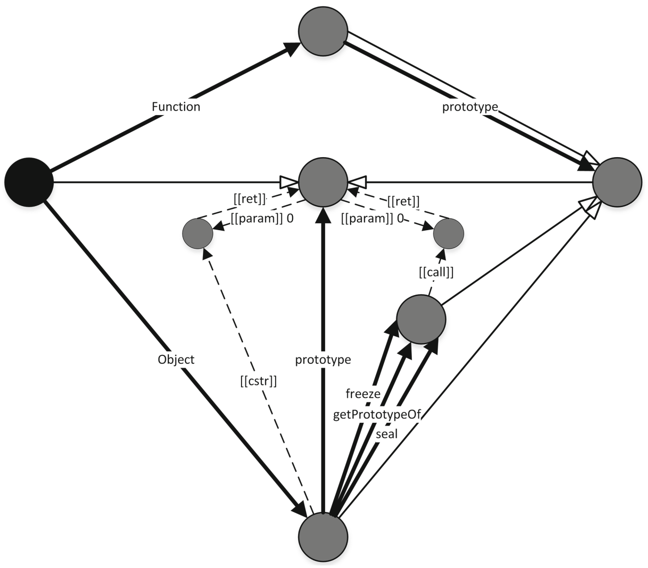

Fig. 1. Example prototype graph manually constructed based on a portion of the ECMAScript 5.1 standard [3, Sects. 15.2, 15.3]. Large and small nodes represent *type* and *sig* vertices respectively. (The single black node on the left represents the type of the global object, however, that is for identification 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.

Informally, a prototype graph is a collection of *type* and *sig* vertices connected by six different kind of edges (and several 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 name information and function argument order is encoded in edge labels. (Note, that vertices have no labeling; most information is stored in the existence of and labels of edges.)

As an example, Fig. 1 shows a prototype graph of 6 *type* and 2 *sig* vertices, manually constructed based on a portion of the ECMAScript 5.1 standard. The graph contains the types of `Object`, `Object.prototype`, `Function`, and `Function.prototype` objects, the global object, and also the constructor signature for `Object` and the call signatures for three functions of it.

Finally, we define some useful notations on prototype graphs as follows.

Definition 2 (Notations). Let $G = \langle V, E, s, t, l_{prop}, l_{param} \rangle$ be a prototype graph, where $V = V_{type} \cup V_{sig}$ and $E = E_{proto} \cup E_{prop} \cup E_{cstr} \cup E_{call} \cup E_{param} \cup E_{ret}$, according to Definition 1. Then we introduce the following notations:

– Let $v \xrightarrow[x]{e} v'$, denote the edge e of type x, where $x \in \{proto, prop, cstr, call, param, ret\}$, iff $e \in E_x \wedge s(e) = v \wedge t(e) = v'$.

– Let $v \xrightarrow[X]{e_1 \dots e_n}{}^{*} v'$, denote the finite path $e_1 \dots e_n$ over X type of edges, where $X \subseteq \{proto, prop, cstr, call, param, ret\}$, iff $e_1 \dots e_n$ is a sequence of edges, with $n \geq 1$, such that $\forall 1 \leq i \leq n : v_i \xrightarrow[x_i]{e_i} v_{i+1} \wedge x_i \in X$, and $v_1 = v \wedge v_{n+1} = v'$.
– Let P_G denote the set of all finite paths in G.

3 Valid API Call Expressions from a Graph Representation

In this section, we present an application of the prototype graph, i.e., JavaScript engine API fuzzing, as motivated in the introduction. Thus, given a built graph, our goal is to generate call expressions that invoke functions on API objects with type-correct arguments. Therefore, we formally define with graph terms how such expressions can look like.

Definition 3 (Function Call Expressions). Let $G = \langle V, E, s, t, l_{prop}, l_{param} \rangle$ be a prototype graph, where $V = V_{type} \cup V_{sig}$, $E = E_{proto} \cup E_{prop} \cup E_{cstr} \cup E_{call} \cup E_{param} \cup E_{ret}$, and $l_{prop} : E_{prop} \to \Sigma$, according to Def. 1. Let $\Lambda : \mathcal{P}(V) \to \mathcal{P}(\Sigma_+^*)$ be a function that maps a set of types to a set of literals from an extended alphabet $\Sigma_+ \supseteq \Sigma \cup \{\boxed{\texttt{new}}, \boxed{\texttt{.}}, \boxed{\texttt{(}}, \boxed{\texttt{,}}, \boxed{\texttt{)}}\}$. Then we define $\Phi_{G,\Lambda} : V_{type} \to \mathcal{P}(\Sigma_+^*)$ as a function that gives for some selected starting vertex v_0 a set of type-correct call expressions that are available from an object of that type:

$$\Phi_{G,\Lambda}(v_0) = \Gamma_{G,\Lambda}(v_0, V_{sig})$$

$\Phi_{G,\Lambda}$ is given with the help of several auxiliary functions, all of which are defined below:

– The function $\Gamma_{G,\Lambda} : V \times \mathcal{P}(V) \to \mathcal{P}(\Sigma_+^*)$ gives the set of type-correct expressions available along a set of paths:

$$\Gamma_{G,\Lambda}(v_0, V') = \bigcup_{e_1 \dots e_n \in \Pi_G(v_0, V')} \left(\left(\boxed{\texttt{new}} \cdot \boxed{\texttt{(}} \right)^{|\{e_i \in E_{cstr}\}|} \cdot \Lambda(\{v_0\}) \cdot \prod_{i=1}^{n} \boxed{\texttt{)}}^{[e_i \in E_{cstr}]} \cdot \gamma_{G,\Lambda}(v_0, e_i) \right)$$

– The function $\gamma_{G,\Lambda} : V \times E \to \mathcal{P}(\Sigma_+^*)$ gives a set of sub-expressions for a step (i.e., edge) along a path:

$$\gamma_{G,\Lambda}(v_0, e) = \begin{cases} \boxed{\texttt{.}} \cdot l_{prop}(e) & \text{if } e \in E_{prop}, \\ \boxed{\texttt{(}} \cdot \left(\prod_{i=1}^{n} {}^{\boxed{\texttt{,}}} \Gamma_{G,\Lambda}(v_0, \nabla_G(v_i)) \cup \Lambda(\nabla_G(v_i)) \right) \cdot \boxed{\texttt{)}} & \\ & \text{if } e \in E_{cstr} \cup E_{call}, \text{ where} \\ & n = |\{e' : \exists v' : v' \xrightarrow[param]{e'} t(e)\}| \text{ and} \\ & \forall 1 \leq i \leq n : v_i \xrightarrow[param]{e_i} t(e) \text{ and} \\ & \forall 1 \leq i < n : l_{param}(e_i) < l_{param}(e_{i+1}), \\ \lambda & \text{otherwise.} \end{cases}$$

- The function $\mathit{\Pi}_G : V \times \mathcal{P}(V) \rightarrow \mathcal{P}(P_G)$ gives the set of all finite paths from a vertex to a set of vertices over *proto*, *prop*, *cstr*, *call*, and *ret* edges:

$$\mathit{\Pi}_G(v, V') = \left\{ e_1 \ldots e_n : v \xrightarrow[\mathit{proto,prop,cstr,call,ret}]{e_1 \ldots e_n}{}^{*} v' \wedge v' \in V' \right\}.$$

- The function $\nabla_G : V \rightarrow \mathcal{P}(V)$ gives the set of all vertices reachable from a given vertex backwards over *proto* edges:

$$\nabla_G(v) = \left\{ v' : \exists e_1 \ldots e_n : v' \xrightarrow[\mathit{proto}]{e_1 \ldots e_n}{}^{*} v \right\}.$$

- The $\cdot$, $\prod$, and power notations denote concatenation of strings (or sets of strings) over an alphabet, and λ is the empty word. The $\prod$ notation with an additional superscript symbol denotes concatenation with a separator symbol:

$$\prod_{i=1}^{n}{}^{\sigma} a_i = \begin{cases} \left(\prod_{i=1}^{n-1}{}^{\sigma} a_i \right) \cdot \sigma \cdot a_n & \text{if } n > 1, \\ \prod_{i=1}^{n} a_i & \text{otherwise.} \end{cases}$$

- $[P]$ denotes the Iverson bracket, i.e., gives 1 if P is true, 0 otherwise.

As visible from the definition above, the graph representation is capable of describing how a property can be accessed (the first case of $\gamma_{G,\Lambda}$), how a function can be parametrized to retrieve a type-correct value (the second case of $\gamma_{G,\Lambda}$), and how a new object can be created with a constructor call (the E_{cstr}-related parts of $\Gamma_{G,\Lambda}$). There is one more way of creating an expression of a given type, however: with literals. Since they fall outside the expressiveness of the graph representation – mostly because a literal is more of a syntactic entity while the concepts in the graph represent components of type information – possible literals of a type (or some types) are represented by the Λ function that complements the graph.

An actual implementation of the above formalism, e.g., a fuzzer, would most probably not generate the (potentially infinite) set $\Phi_{G,\Lambda}(v_0)$ but may choose an arbitrary element from it (and perhaps a different one on every execution). That can be achieved by a random walk on the prototype graph based on the informal concept behind the formal definition: "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..." Also, in an API fuzzer, v_0 would most probably be the type of global object, and Λ would map v_0 to a literal referring to the global object.

As an example, below we give some expressions in $\Phi_{G_{ex},\Lambda_{ex}}(v_{ex})$, where G_{ex} is the graph in Fig. 1, v_{ex} is the type of the global object in that graph (marked with black), and $\Lambda_{ex}(V') = \bigcup_{i=1}^{[v_{ex} \in V']} \{ \boxed{\texttt{this}} \}$:

- `this.Object.getPrototypeOf(this.Function.prototype)`,
- `new (this.Object)(this)`.

4 Building a Prototype Graph

The most trivial way of building a prototype graph may seem to be by hand. However, processing web standards manually – not only that of ECMAScript but potentially others as well, e.g., HTML5 DOM bindings – may require huge human efforts and is still heavily error prone, will be hard to maintain, and will not know about any engine-specific extensions. Thus, we outline two automated techniques below.

Both approaches rely on the introspecting capabilities of the JavaScript language: not only can we determine the actual type of every expression at runtime (with the `typeof` operator, e.g., whether it is of primitive boolean, number, or string type, or an object or a function, to name the most important types) but we can enumerate all properties and walk the property chains of all objects (with `Object.getOwnPropertyNames()` and `Object.getPrototypeOf()`), and retrieve the number of the formal parameters of every function (by reading their `length` property). By relying on introspection, the approaches do not require the source code of the engine-to-be-tested making them applicable to any engines.

The first approach, which we will call *engine discovery* in the paper (or just discovery, for short), is a one-time analysis of the engine. It's core idea is to execute a specially crafted JavaScript program inside the engine-to-be-tested using the above mentioned constructs. If this JavaScript program gets access to an object, it can create type descriptor data structures from all values recursively reachable from it by recording property names, and prototype and property relations – and formal argument list lengths for function objects. If this JavaScript program gets access to the global object of the execution environment then it can discover the whole API of the engine. Fortunately, the language standard mandates the existence of that global object (available as `this` in the outmost lexical scope), thus the approach is universal for all engines.

The discovery technique can find the prototypes and properties in the exposed API, but as mentioned above, it has a very limited view of the signatures (parameter and return types) of the functions. Therefore, we propose a second approach to extend the information about functions in the type descriptor data structure: in JavaScript, every object can be modified dynamically, even most of the built-in objects, and this includes the replacement of built-in functions with user-written JavaScript code. Thus, we propose to wrap (or patch) every function found using the discovery technique in a special code that, when called, collects type descriptor information about every parameter before passing them through to the original function, and also collects details about the returned value after the original call finished but before it is given back to the caller. This way the observable behaviour of the wrapped system does not change, but as programs execute in the engine, information about function signatures in the type descriptors can be continuously extended, refined (or, *learned*). Executing official or project-specific JavaScript test suites in such a patched environment is a plausible choice to learn signatures.

The data structures built by the engine discovery and signature learning approaches can be exported from the engine, e.g., in JSON format by using

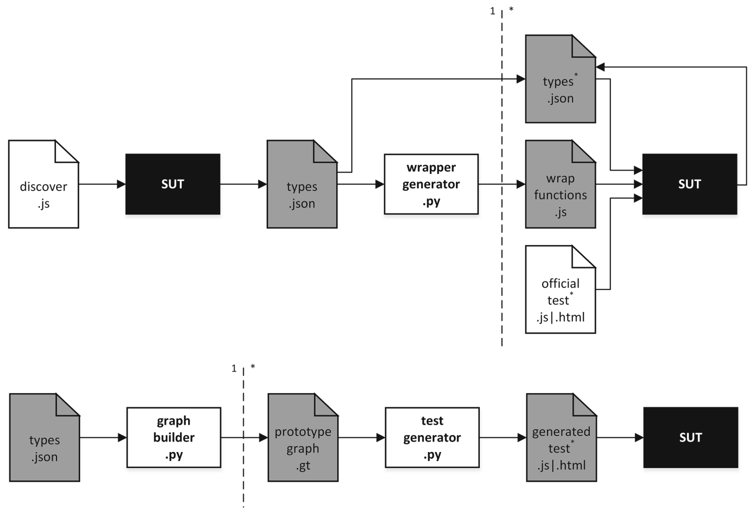

Fig. 2. Architecture overview of the prototype implementation of the graph-based JavaScript engine API fuzzing technique. The black boxes stand for the SUT, white elements are part of the implementation, while gray ones are generated during its execution. Dashed lines separate parts which execute only once from those which run multiple times (changing elements are marked with *).

built-in conversion routines. Then, that information can be easily transformed to the prototype graph format introduced in Sect. 2.

5 Experimental Results

5.1 Tools and Environment

To evaluate the ideas explained in the previous sections, we have created a prototype implementation. The code that discovers the API of the engine and learns signatures from existing tests was written in JavaScript (relying on the introspection capabilities of the language, as described in Sect. 4), while test generator code, execution harness for the engine-under-test, and utility routines were implemented in Python 3, with the help of the graph-tool module[5]. The architecture overview of the prototype implementation is shown in Fig. 2.

As SUT, we have chosen *jsc*, the command line JavaScript execution tool from the WebKit project[6]. The project – checked out from its official code repository at revision 192323 dated 2015-11-11 – was built in debug configuration for its GTK+ port (i.e., external dependencies like UI elements were provided by the GTK+ project[7]), on an x86-64 machine running Ubuntu 14.04 with Linux kernel 3.13.0, with gcc 4.9.2. To enable the investigation of the effects of fuzzing

[5] https://graph-tool.skewed.de/.
[6] https://webkit.org/.
[7] http://www.gtk.org/.

on the SUT, we made use of the coverage computation support of the compiler together with the gcov 4.9.2 and gcovr 3.2 tools[8].

Our goal was not only to evaluate the prototype graph-based fuzzing technique on its own but also to compare it to existing solutions. Therefore, we have downloaded jsfunfuzz[9] – with hash 6952e0f dated 2016-01-08 – , one of the few available open source JavaScript fuzzers to act as a baseline for our experiments.

5.2 Graphs

Since in our fuzzing technique the first step is to build a prototype graph, we have applied both previously outlined approaches to *jsc*, i.e., first we have discovered the engine and then learned functions signatures from existing tests. For the latter step, we have used the official JavaScript stress test suite of the WebKit project, executing 3,573 tests in the wrapped engine.

Table 1 shows the size metrics of the built graphs. The most striking difference is in the number of V_{type} vertices. This can be attributed to two factors: first, because of the limited view of the engine discovery on the function signatures, a large number of function objects are considered to be of the same prototype (i.e., only their argument number differentiates between them, and those with the same number of arguments get represented by a single type vertex). With signature learning, however, most of the function objects get represented by a separate type vertex. The second factor in the increase are new types emerging during signature learning either originating from the test suite (e.g., previously unseen literal parameters) or from the engine itself (e.g., natively constructed return values).

Figure 3 shows the prototype graph resulting after the learning step.

Table 1. Size metrics of prototype graphs built for *jsc*. (Signatures were learned from 3,573 official stress tests.)

	pgdiscover	pglearn
$\|V_{type}\|$	90	2022
$\|V_{sig}\|$	6	2362
$\|E_{proto}\|$	83	779
$\|E_{prop}\|$	670	2297
$\|E_{cstr}\|$	0	473
$\|E_{call}\|$	32	2404
$\|E_{param}\|$	17	83300
$\|E_{ret}\|$	6	2362

[8] http://gcovr.com/.
[9] https://github.com/MozillaSecurity/funfuzz/.

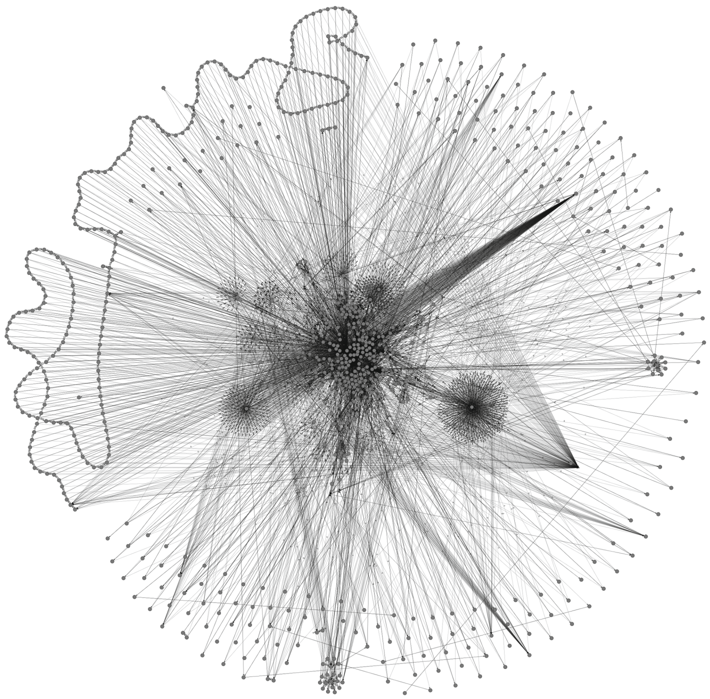

Fig. 3. Prototype graph built for *jsc* with the signature learning technique.

5.3 Evaluation

Once the graphs were built, we have used our test generator implementation to create 50,000 expressions from both, each expression resulting from a path consisting of a maximum of 20 property steps (as defined by $\gamma_{G,\Lambda}$ in Sect. 3). Then, we executed all expressions in *jsc*, one by one, and monitored their results. We took note of all assertion failures and program crashes, and after every 5,000 expressions we also measured the accumulated code coverage of the hitherto executed tests.

For the execution of jsfunfuzz, we used its own framework with the slight modification of adding a periodic code coverage measurement to the system. Jsfunfuzz was written with continuous (or continuously restarting) execution in mind and thus does not support the generation of a given number of expressions. Therefore, we could neither generate one expression per test as with our implementation nor could we execute exactly 50,000 expressions. However, we wanted to keep changes to it to the minimum, therefore we did not alter that behavior. To allow fair comparison, we parametrized jsfunfuzz to restart after every 1 second to generate only a small amount of expressions per test and we stopped the whole framework after the first code coverage measurement past the 50,000 expressions limit (thus finishing with 50,320 expressions).

Table 2 shows the line coverage results of all three fuzzing approaches both with module-level granularity and in total. (We have considered each subdirectory and standalone file of the *JavaScriptCore* build folder of the WebKit project as a separate module.) The results show that the basic engine discovery-based approach does not perform as well as jsfunfuzz in terms of total code coverage (23.31 % compared to 37.25 %), but as soon as we extend our graph with signature information extracted from tests, it improves significantly and gives higher results (44.13 %).

We should also highlight results on three important modules, namely on *runtime*, *yarr*, and *jsc.cpp*. The first contains C++ implementations of core JavaScript language functionality (like built-in functions), while the second contains the regular expression engine of the project. The third module (actually, a single file), is the main command line application, which is a classic JavaScript engine embedder in the sense that it binds some extra, non-standard routines into the JavaScript environment. That is, these are the modules that expose API of native code to the JavaScript space and thus are in our focus. As the table shows, even the simpler engine discovery-based technique can outperform jsfunfuzz in two out of the three modules, and the signature-extended variant gives the best results in all three cases.

For the sake of completeness, we should also mention some modules where the coverage is very low for all techniques. *API* and *bindings* contain API exposed to an embedder: a JavaScript code has no effect on how the engine is linked to its container application (moreover, *bindings* is marked deprecated in the code base, thus won't reach a higher coverage ever). The other modules – i.e., *debugger*, *disassembler*, *inspector*, *profiler*, and *tools* – contain code that are development-related and are also not under the control of the JavaScript code. Thus, these modules cannot be – and are not – the target of JavaScript API fuzzing.

As the ultimate goal of fuzzing is not only to reach good code coverage but also to cause system malfunction, we compared the three techniques on the basis of caused crashes as well. Table 3 shows the total number of observed failures, and since several tests triggered the same problem, the number of unique failures as well. (The uniqueness was determined based on crash backtrace information retrieved with a debugger.) Interestingly, both graph-based fuzzing techniques found the same failures. This also means that even the engine discovery technique with lower total coverage ratio could find more errors than jsfunfuzz. (However, it has to be noted that one of the two crashes caused by jsfunfuzz was not found by the graph-based prototype implementation.)

6 Related Work

Several previous authors tried to handle the weak-typedness of the JavaScript language by creating various type systems. One of the most well-known type systems was introduced by Anderson [1], who gave a formal algebraic definition for types of a language named JS_0 and also described how to perform type inference on such grammars. Other authors extended that work, like Franzen

Table 2. Code coverage results on *jsc* after 50,000 generated expressions (after 50,320 expressions for jsfunfuzz).

Module	Total lines	Covered lines					
		pgdiscover		pglearn		jsfunfuzz	
API	1698	9	0.53 %	9	0.53 %	9	0.53 %
DerivedSources	4546	148	3.26 %	167	3.67 %	312	6.86 %
assembler	2997	1046	34.90 %	2037	67.97 %	2054	68.54 %
bindings	165	0	0.00 %	0	0.00 %	0	0.00 %
builtins	96	63	65.63 %	63	65.63 %	62	64.58 %
bytecode	8578	1650	19.24 %	4196	48.92 %	3320	38.70 %
bytecompiler	4656	2344	50.34 %	2372	50.95 %	2887	62.01 %
debugger	713	3	0.42 %	3	0.42 %	3	0.42 %
dfg	29959	27	0.09 %	11019	36.78 %	9403	31.39 %
disassembler	1033	3	0.29 %	3	0.29 %	3	0.29 %
heap	4221	2517	59.63 %	2671	63.28 %	2373	56.22 %
inspector	3594	0	0.00 %	0	0.00 %	0	0.00 %
interpreter	1336	594	44.46 %	664	49.70 %	648	48.50 %
jit	8919	814	9.13 %	4852	54.40 %	4345	48.72 %
jsc.cpp	926	507	54.75 %	519	56.05 %	240	25.92 %
llint	840	344	40.95 %	424	50.48 %	451	53.69 %
parser	6586	3618	54.93 %	3801	57.71 %	4400	66.81 %
profiler	788	4	0.51 %	4	0.51 %	4	0.51 %
runtime	27112	12115	44.69 %	15101	55.70 %	10648	39.27 %
tools	534	13	2.43 %	13	2.43 %	13	2.43 %
yarr	3538	486	13.74 %	1879	53.11 %	856	24.19 %
TOTAL	112835	26305	23.31 %	49797	44.13 %	42031	37.25 %

Table 3. Number of failures caused in *jsc*.

	pgdiscover	pglearn	jsfunfuzz
total failures	1326	1445	4
unique failures	6	6	2

and Aspinall [4], who also tried to reason about the resource usage of programs with the help of the type system. However, even if designed to be "realistic", JS_0 is only a subset of the complete JavaScript language to make it manageable with respect to formalization.

Other type system and type inference approaches have also been proposed [2,7,13] but all authors focused on the static analysis of applications, not on the API of engines or on the execution engines themselves. Sen et al. have created

Jalangi [10], a dynamic analysis framework for JavaScript but it also deals with user code only with the help of preprocessing and not with the environment the programs run in, just like the static analyses.

The origins of random test generation date back to at least the '70s, when Purdom published his paper about grammar-based sentence generation for parser testing [9]. However, the term fuzzing was coined by Miller in 1988 only, as a result of truly random noise appearing in modem lines, disturbing terminals, and causing programs to crash [12]. Since then, randomized testing has become widespread; a good overview is also given in the work of Sutton et al [11].

The records on JavaScript fuzzing are not that long, but as the importance of the language started raising, the topic gained attention both in academia and industry. Godefroid et al. [5] have presented a whitebox fuzzing technique and experimented with it on the JavaScript interpreter of the Internet Explorer 7. That approach, however, required the symbolic execution of the application and constraint solving in addition. Holler et al. created LangFuzz [6] but that is an almost purely syntax-directed approach, which aims to avoid introducing language-dependent semantic knowledge into the fuzzer, thus LangFuzz has no type representation at all. Closest to our work is the state-of-the-art jsfunfuzz system from Ruderman, which we have chosen to compare our proposed technique against. That system does apply an engine discovery approach similar to ours, but it does not utilize all introspecting possibilities of the language, e.g., does not traverse the prototype chains and completely omits the automated discovery and learning of function signatures (all possible parametrizations are manually specified for the fuzzer).

7 Summary and Future Work

In this paper, we have defined a graph-based formalization of type information in JavaScript, we have shown how to automate the building of such graphs to describe the API exposed by a JavaScript execution engine, and we have also defined an API fuzzing method based on graph terms. According to our knowledge, this paper is the first work to use graph formalization and traversal for type information and related analyses of JavaScript. In addition to the formal definitions, we have also presented the experimental results of a prototype implementation of a JavaScript engine API fuzzer. The results show that the prototype graph based API fuzzing technique can be on par with available solutions in terms of total code coverage and even yield significantly better coverage results in JavaScript API-related modules. Moreover, the implementation triggered real program crashes during the experiments.

We see several potential directions for future work. First of all, we would like to enhance the potential of the API fuzzer. We plan to experiment with changing the formulas of Sect. 3 so that they do not necessarily generate type-correct expressions (e.g., by replacing the occurrences of ∇_G with a function that traverses *proto* edges in both directions instead of forward only). That may (or probably will) cause more generated inputs to be discarded by the engines, but we speculate that it may trigger more intrigued bugs as well. We would also like to evaluate our approach on other SUTs: both on different standalone JavaScript engines and especially on systems that bind external APIs into the JavaScript execution environment, e.g., on web browsers that expose HTML DOM manipulation API to user code. Moreover, we plan to investigate the recently published ECMAScript 6 standard and evolve both the formalizations and the implementation to adapt to any new concepts as needed. Finally, we plan to investigate the potential use cases of the prototype graph outside the fuzzing domain as well.

References

1. Anderson, C.L.: Type inference for JavaScript. Ph.D. thesis, University of London, Imperial College London, Department of Computing (2006)
2. Chugh, R., Herman, D., Jhala, R.: Dependent types for JavaScript. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2012), pp. 587–606. ACM (2012)
3. Ecma International: ECMAScript Language Specification (ECMA-262), 5.1st edn., June 2011
4. Franzen, D., Aspinall, D.: Towards an amortized type system for JavaScript. In: Proceedings of the 6th International Symposium on Symbolic Computation in Software Science (SCSS 2014). EPiC Series in Computer Science, vol. 30, pp. 12–26. EasyChair (2014)
5. 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)
6. Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: 21st USENIX Security Symposium, pp. 445–458. USENIX (2012)
7. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009)
8. Microsoft Corporation: Security development lifecycle (verification phase). https://www.microsoft.com/en-us/sdl/default.aspx
9. Purdom, P.: A sentence generator for testing parsers. BIT Numer. Math. **12**(3), 366–375 (1972)
10. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In: Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013), pp. 488–498. ACM (2013)
11. Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley, Boston (2007)
12. Takanen, A., DeMott, J., Miller, C.: Fuzzing for Software Security Testing and Quality Assurance, chap. Foreword, Artech House (2008)
13. Thiemann, P.: Towards a type system for analyzing JavaScript programs. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 408–422. Springer, Heidelberg (2005)