JSForce: A Forced Execution Engine for Malicious JavaScript Detection

Xunchao Hu[1](✉), Yao Cheng[1], Yue Duan[2], Andrew Henderson[1], and Heng Yin[2]

[1] Department of EECS, Syracuse University, Syracuse, USA
{xhu31,ycheng}@syr.edu, hendersa@icculus.org
[2] Department of Computer Science and Engineering, University of California, Riverside, Riverside, USA
yduan005@ucr.edu, heng@cs.ucr.edu

Abstract. The drastic increase of JavaScript exploitation attacks has led to a strong interest in developing techniques to analyze malicious JavaScript. Existing analysis techniques fall into two general categories: static analysis and dynamic analysis. Static analysis tends to produce inaccurate results (both false positive and false negative) and is vulnerable to a wide series of obfuscation techniques. Thus, dynamic analysis is constantly gaining popularity for exposing the typical features of malicious JavaScript. However, existing dynamic analysis techniques possess limitations such as limited code coverage and incomplete environment setup, leaving a broad attack surface for evading the detection. To overcome these limitations, we present the design and implementation of a novel JavaScript forced execution engine named `JSForce` which drives an arbitrary JavaScript snippet to execute along different paths without any input or environment setup. We evaluate `JSForce` using 220,587 HTML and 23,509 PDF real-world samples. Experimental results show that by adopting our forced execution engine, the malicious JavaScript detection rate can be substantially boosted by 206.29% using same detection policy without any noticeable false positive increase.

Keywords: Malicious Javascript · Forced execution

1 Introduction

Malicious JavaScript has become an important attack vector for software exploitation attacks. According to a recent report from Symantec [3], there are millions of victims attacked by malicious JavaScript on the Internet each day. A number of techniques [7–9,12–14,18] have been proposed to detect malicious JavaScript code. Due to the dynamic features of the JavaScript language, static analysis [9,10] can be easily evaded using obfuscation techniques [24]. Consequently, researchers rely upon dynamic analysis [8,11,14] to expose the typical

A full version of this paper can be found at https://arxiv.org/abs/1701.07860.

features of malicious JavaScript. More specifically, these approaches rely on visiting websites or opening PDF files with a full-fledged or emulated browser/PDF reader and then monitoring the different features (e.g., heap health [18].) for detection.

However, the typical JavaScript malware is designed to execute within a particular environment, since they aim to exploit specific vulnerabilities, as opposed to benign JavaScript, which will run in a more environment-independent fashion. Fingerprinting techniques [22] are widely adopted by JavaScript malware to examine the runtime environment. A dynamic analysis system may fail to observe some malicious behaviors if the runtime environment is not configured as expected. Such configuration is quite challenging because of the numerous possible runtime environment settings. Hence, existing dynamic analysis systems usually share the limitations of limited code coverage and incomplete runtime environment setup, which leave attackers with a broad attack surface to evade the analysis.

To solve those limitations, we propose `JSForce`, a forced execution engine for JavaScript, which drives an arbitrary JavaScript snippet to execute along different paths without any input or environment setup. While increasing code coverage, `JSForce` can tolerate invalid object accesses while introducing no runtime errors during execution. This overcomes the limitations of current JavaScript dynamic analysis techniques. Note that, as an amplifier technique, `JSForce` does not rely on any predefined profile information or full- fledged hosting programs like browsers or PDF viewers, and it can examine partial JavaScript snippets collected during an attack. As demonstrated in Sect. 4, `JSForce` can be leveraged to improve the detection rate of other dynamic analysis systems without modification of their detection policies. While the high-level concept of forced execution has been introduced in binary code analysis (X-Force [17]), we face unique challenges in realizing this concept in JavaScript analysis, given that JavaScript and native code are very different languages by nature.

We implement `JSForce` on top of the V8 JavaScript engine [5] and evaluate the effectiveness, and runtime performance of `JSForce` with 220,587 HTML files and 23,509 PDF samples. Our experimental results demonstrate that adopting `JSForce` can greatly improve the JavaScript analysis results by 206.29% without any noticeable increase in false positives and with reasonable performance overhead.

Our main contributions are summarized as follows:

(1) We propose JavaScript forced execution technique that forces a JavaScript snippet to execute along different paths while requiring no inputs or any environment setup, to overcome the current limitations of existing JavaScript dynamic analysis techniques: limited code coverage and incomplete runtime environment setup.
(2) To enable forced execution of JavaScript, we develop a type inference model to detect and properly recover from exceptions. We have also developed path exploration algorithms for malicious JavaScript code analysis.

(3) We implement the technique with a prototype system, named `JSForce`, and evaluate its effectiveness, and runtime performance. Experimental results show that by adopting `JSForce`, the malicious JavaScript detection rate is substantially increased by 206.29% while still using the same detection policy. This increase comes without any noticeable increase in false positives and with runtime performance that is very suitable for large-scale analysis.

2 Related Work and Overview

Malicious JavaScript Code. Malicious JavaScript code is typically obfuscated and will attempt to fingerprint the version of the victim's software (browser, PDF reader, etc.), identify vulnerabilities within that software or the plugins that software uses, and then launch one or more exploits. Figure 1 shows a listing of JavaScript code used for a drive-by-download attack against the Internet Explorer browser. Line 1 employs precise fingerprinting to deliver only selected exploits that are most likely to attack the browser. Lines 5–7 contain evasive code to bypass emulation-based detection systems. More precisely, the code attempts to load a non-existant ActiveX control, named `UM0QS4dD` (line 6). When executed within a regular browser, this operation fails, triggering the execution of the `catch` block that contains the exploitation code (lines 7–14).

```
1  if ((navigator.appName.indexOf(”Microsoft Inte” + ”rnet Explorer”
     ) == 1) && (navigator.userAgent.indexOf(”Windows N” + ”T 5.1”) ==
     1) && (navigator.userAgent.indexOf(”MSI” + ”E 8.0”) == 1))
     {
2  att = btt + 1;
3  }
4  if (att == 0) {
5  try {
6  new ActiveXObject(”UM0QS4dD”);
7  } catch (e) {
8  var tlMoOul8 = '\x25' + 'u9' + '\x30' + '\x39' + YYGRl6;
9  tlMoOul8 += tlMoOul8;
10 var CBmH8 = ”%u”;
11 var vBYG0 = unescape;
12 var EuhV2 = ”BODY”;
13 ...
14 }
15 }
16 setTimeout(”redir()”, 3000);
```

Fig. 1. The Malicious JavaScript sample

Types:

$$\tau \quad ::= \sum_{i \in T, T \subseteq \{\perp, u, b, s, n, o\}} \varphi_i$$

Rows:

$$\begin{array}{lcl} \varrho & ::= & str{:}\tau, \varrho \\ & | & \varrho_\tau \end{array}$$

Type environments:

$$\begin{array}{lcl} \Gamma & ::= & \Gamma(x : \tau) \\ & | & \varnothing \end{array}$$

Type summands and indices:

$$\begin{array}{lcl} \varphi_\perp & ::= & \text{Undef} \\ \varphi_u & ::= & \text{Null} \\ \varphi_b & ::= & \text{Bool}(\xi_b) \\ \xi_b & ::= & \mathit{false} \mid \mathit{true} \mid \top \\ \varphi_s & ::= & \text{String}(\xi_s) \\ \xi_s & ::= & \mathit{str} \mid \top \\ \varphi_n & ::= & \text{Number}(\xi_n) \\ \xi_n & ::= & \mathit{num} \mid \top \\ \varphi_f & ::= & \text{Function}(\mathit{this} : \tau; \varrho \to \tau) \\ \varphi_o & ::= & \text{Obj}(\sum_{i \in T, T \subseteq \{b,s,n,f,\perp\}} \varphi_i)(\varrho) \\ \varphi_{fo} & ::= & \text{FObj} \\ \varphi_{ff} & ::= & \text{FFun} \end{array}$$

Fig. 2. Syntax of JavaScript types

However, an emulation-based detection system must emulate the ActiveX API by simulating the loading and presence of any ActiveX control. In these systems, the loading of the ActiveX control will not raise this exception. As a result, the execution of the exploit never occurs and no malicious activity is observed. Instead, the victim is redirected to a benign page (line 16) if the fingerprinting or evasion stage fails. Attackers can also abuse the function `setTimeout` to create a time bomb [6] to evade detection. Detection systems can not afford to wait for long periods of time during the analysis of each sample in an attempt to capture randomly triggered exploits.

Challenges and Existing Techniques. Static analysis is a powerful technique that explores all paths of execution. But, one particular issue that plagues static analysis of malicious JavaScript is that not all of the code can be statically observed. For example, static analysis cannot observe malicious code hidden within `eval` strings, which are frequently exploited by attackers to obfuscate their code. Therefore, current detection approaches [8,11,14] rely upon dynamic analysis to expose features typically seen within malicious JavaScript. More specifically, these approaches rely upon visiting websites or opening PDF files with an instrumented browser or PDF reader, and then monitoring different features (`eval` strings [11], heap health [18], etc.) for detection.

However, dynamic analysis techniques suffer from two fundamental limitations. The first limitation is limited code coverage. This becomes a much more severe limitation within the context of analyzing malicious JavaScript. Attackers frequently employ the *cloaking* [23] technique, which works by fingerprinting the victim's web browser and only revealing the malicious content when the victim is using a specific version of the browser with a vulnerable plugin. Cloaking makes dynamic analysis much harder because the sample must be run within every combination of web browser and plugin to ensure complete code coverage. The widely-used event callback feature of JavaScript also makes it challenging for dynamic analysis to automatically trigger code. For example, attackers can load the attack code only when a specific mouse click event is captured, and automatically determining and generating such a trigger event is difficult.

The second limitation is the complexity of the JavaScript runtime environment. JavaScript is used within many applications, and it can call the functionality of any plugin extensions supported by these applications. For dynamic analysis, any pre-defined browser setup handles a known set of browsers and plugins. Thus, there is no guarantee that this setup will detect vulnerabilities only present in less popular plugins. While it is possible to deploy a cluster of machines running many different operating systems, browser applications, and browser plugins, the exponential growth of possible combinations rapidly causes scalability issues and makes this approach infeasible.

Rozzle [13] attempts to address this code coverage problem by exploring environment-related paths within a single execution. For instance, because `att` in Fig. 1 depends upon the environment-related API's output, Rozzle will execute lines 5–15 and reveal the malicious behaviors hidden in lines 8–14 by executing both the `try` and `catch` blocks. But, it requires a predefined environment-related profile for path exploration. Construction of a complete profile is a challenging task because of the numerous different browsers and plugins, especially for newer proposed fingerprinting techniques [15,16,22]. These new techniques do not rely upon any specific APIs. For instance, the JavaScript engine fingerprinting technique [16] relies upon JavaScript conformance tests such as the Sputnik [4] test suite to determine a specific browser and major version number. There are no specific APIs used for the fingerprinting. Thus, Rozzle cannot include it within the predefined profile and explore the environment-related paths. Rozzle also introduces runtime errors into the analysis engine, which may stop the analysis before any malicious code is executed. In contrast, `JSForce` does not rely upon predefined profile for path exploration and handles runtime errors using the forced execution model presented in Sect. 3.1. By overcoming those limitations of Rozzle, `JSForce` achieves greater code coverage.

Revolver [12] employs a machine learning-based detection algorithm to identify evasive JavaScript malware. However, it requires that the malicious sample is present within a known sample set so that its evasive version can be determined based upon the classification difference. By design, it can not be used for 0-day malware detection.

Symbolic execution has also been applied to the task of exposing malware [6]. This technique, while improving code coverage over dynamic analysis, suffers from scalability challenges and is, in many ways, unnecessarily precise [13]. Within the context of JavaScript analysis, symbolic execution becomes more challenging [19]. JavaScript applications accept many different kinds of input, and those inputs are structured as strings. For example, a typical application might take user input from form fields, messages from a server via `XMLHttpRequest`, and data from code running concurrently within other browser windows. It is extremely difficult for a symbolic string solver [21] to effectively supply values for all of these different kinds of inputs and reason about how those inputs are parsed and validated. The rapidly evolving JavaScript language and its host programs (browsers, PDF readers, etc.) make the modeling of the JavaScript API tedious work. Furthermore, the dynamic features (such as the `eval` function) of JavaScript make symbolic execution infeasible for many analysis efforts.

Overview. `JSForce`, our proposed forced-execution engine for JavaScript, is an enhancement technology designed to better expose the behaviors of malicious JavaScript at runtime. Different detection policies can be applied to examine malicious JavaScript. While the forced execution concept is first introduced for binary code analysis (X-Force [17]), we face unique challenges, such as type inference and invalid object access recovery, in enabling the forced execution concept for JavaScript.

We now illustrate how the forced execution of JavaScript code works. Consider the snippet shown in Fig. 1. `JSForce` forces the execution through the different code paths of the snippet. So, the exploitation code within the `catch` block (lines 7–14) will be executed, no matter how the ActiveX API is simulated by the emulation-based analysis system. Moreover, `JSForce` will immediately invoke the callback function passed to `setTimeout` to trigger the time bomb malware.

`JSForce`'s path exploration forces line 2 to be executed, regardless of the result of the fingerprinting statement (line 1). Since `btt` is not defined within the code snippet under analysis, which is a common scenario because collected JavaScript code may be incomplete due to multi-stages of the attack, the execution of line 2 raises a `ReferenceError` exception when running within a normal JavaScript engine. When the exception is captured, `JSForce` creates a `FakedObject` named `btt`, which is fed to the JavaScript engine to recover from the invalid object access. However, the type of `btt` is unknown at the time of `FakedObject`'s creation. `JSForce` infers the type based upon how the `FakedObject` is used. For example, if this `FakedObject` is added to an integer, `JSForce` will then change its type from `FakedObject` to `Integer`. We call this *faked object retyping*.

3 JavaScript Forced Execution

This section explains the basics of how a single forced execution proceeds. The goal is to have a non-crashable execution. We first present the JavaScript language semantics and then focus on how to detect and recover from invalid object accesses. We then discuss how path exploration occurs during forced execution.

3.1 Forced Execution Semantics

The JavaScript Language. JavaScript is a high-level, dynamic, untyped, and interpreted programming language. At runtime, the JavaScript engine dynamically interprets Java-Script code to (1) load/allocate objects, (2) determine the types of objects, and (3) execute the corresponding semantics. Given an arbitrary JavaScript snippet, execution may fail because of undefined/uninitialized objects or incorrect object types. For instance, the execution of line 2 in Fig. 1 raises a `ReferenceError` exception because `btt` is not defined. To tolerate that, forced execution must handle such failures.

The basic idea behind forced execution is that, whenever a reference error is discovered, a `FakedObject` is created and returned as the pointer of the property. During the execution of the program, the expected type of the `FakedObject` is indicated by the involved operation. For instance, adding a `number` object to a `FakedObject` indicates that the `FakedObject`'s type is `number`. When the type of a `FakedObject` can be determined, we update it to the corresponding type.

Potentially, we could assign `FakedObject` with the type `Object` and reuse the dynamic typing rules of the JavaScript engine to coerce the `FakedObject` to an expected type. Nevertheless, the dynamic typing rules of the JavaScript engine are designed to maintain the correctness of JavaScript semantics and do not suffice to meet our analysis goal of achieving maximized execution.

This can be attributed to two reasons. First, while the JavaScript engine can cast the `FakedObject:Object` to proper primitive values, it cannot cast the `FakedObject:Object` to proper object types. For instance, when a `FakedObject` with the type `Object` is used as a function object, the JavaScript engine will raise the `TypeError` exception according to ECMA specification [1]. Second, the casting of `FakedObject` to primitive values by the JavaScript engine can lead to unnecessary loss of precision. To understand why, consider the following loop:

```
c = a/2;
for(i= c;i<10000;i++)
  memory[i] = nop + nop + shellcode;
```

Since `a` is not defined, a `FakedObject` will be created. With the built-in typing rule of the JavaScript engine, `c` will be assigned the value `NaN`. The loop condition i<10000 will always evaluate to false. Thus, the loop body, which contains the heap spray code, will never be executed. Although the path exploration of `JSForce` will guarantee that the loop body will be executed once, without executing the loop 10,000 times, it will likely be missed by heap spray detection tools because of the small chunk of memory allocated on the heap.

Therefore, to overcome the above two issues, `JSForce` introduces two new types, `FObj` and `FFun`, to the JavaScript type system. The JavaScript type system defined in [20] is extended to support these two new types. Figure 2 summarizes the new syntax of these JavaScript types. Type `FObj` is for `FakedObject`. At the moment `FakedObject` is created, we assign type `FObj` as the temporary type of `FakedObject`. It can be subtyped to any types within the JavaScript type system. When `FakedObject` is used as a function object, `FakedObject` is casted to `FakedFunction` with type `FFun`. The `FakedFunction` with type `FFun` can take arbitrary input and always returns `FakedObject:FObj`. Following `JSForce`'s dynamic typing rules, `a` in the above `loop` sample will be typed to `Number` because it is used as a dividend. `c` is then assigned to `Number` and the loop body is executed repeatedly until the loop condition `i < 10000` is evaluated to false. By introducing these two new types and their typing rules, `JSForce` solves the two issues mentioned in the above paragraph. In the following paragraphs, we detail the JavaScript forced execution model.

Reference Error Recovery. To avoid `ReferenceError` exceptions, we introduce the `FakedObject` and recover the error by creating the `FakedObject` whenever necessary. There are two cases that lead to reference errors. The first case (ER_1) is a failed object lookup. Every field access or prototype access triggers a dynamic lookup using the field or prototype's name as the key. If no object is found, the lookup fails. Such failures happen when the running environment is incomplete or some portion of the JavaScript code is missing. For example, a browser plugin referenced by the JavaScript is not installed, or only a portion of the JavaScript code is captured during the attack (Fig. 4).

To handle this error, `JSForce` intercepts the lookup process and a `FakedObject` named as the lookup key is created whenever a failed lookup is captured. The corresponding parent object's property is also updated to the

```
1  var a = null;
2  var b = c + 1;
3  var d = a.length;
4  var func = null;
5  a = "Hello World";
6  var e = new abc();
7  if (b < 5) {
8      func = function(x)
       {
9          return x
10     };
11 }
12 d = func(6);
13 var f = Math.abs(d);
14 array[5] = f;
```

Fig. 3. JavaScript sample

Statement	Action	Rule
1: var a = null;	$a \mapsfrom FakedObject$	ER_2
2: var b = c + 1;	$c \mapsfrom FakedObject$	ER_1
	$c \mapsfrom RanNumber$	R_BINOPERATOR1
3: var d = a.length;	$a.length \mapsfrom FakedObject$	ER_1
4: var func = null;	$func \mapsfrom FakedObject$	ER_2
5: a = "Hello World";	$a \mapsfrom "HelloWorld"$	R_ASSIGN
6: var e = new abc();	$abc \mapsfrom FakedObject$	ER_1
	$abc \mapsfrom fakedFunction$	R_NEW
7: if(b <5)	NO ACTION	NONE
12: d = func(6)	$func \mapsfrom fakedFunction$	R_CALL1
	$d \mapsfrom FakedObject$	R_ASSIGN
13: var f = Math.abs(d)	$d \mapsfrom RanNumber$	R_CALL2
14: array[5] = f;	$array \mapsfrom FakedObject$	ER_1
	$array \mapsfrom arrayObject$	R_INDEX1
	$array[5] \mapsfrom f$	R_ASSIGN

Fig. 4. Forced execution of sample in Fig. 3

`FakedObject`. Line 2 in Fig. 3 presents such an example. The JavaScript engine searches the current code scope for the definition of `c`, which is not defined. `JSForce` returns the `FakedObject` as the temporary value of `c` so that the execution can continue.

The second case (ER_2) occurs when the object is initialized to the value `null` or `undefined`, but later has its properties accessed. `JSForce` modifies the initialization process to replace the `null` to a `FakedObject` if an object is initialized as value `null` or `undefined`. For example, the variable `a` defined on line 1 in Fig. 3 is assigned the value `FakedObject` instead of `null` under the forced execution engine. The variable `a` may later be updated to another value during execution, but this does not sabotage the execution of JavaScript code.

Faked Object Retyping. When a `FakedObject` is used within an expression, it must be retyped to the expected type. Otherwise, incorrect typing raises a `TypeError` exception and stops the execution. `JSForce` infers the expected type of `FakedObject` by how the `FakedObject` is used. Figure 5 summarizes the dynamic typing rules introduced by `JSForce`. The rules are divided into the following five categories:

(1) *R-ASSIGN.* This rule deals with assignment statements. When a `FakedObject` e_0 is assigned to a new value e_1, e_0 is updated to the new value $e1$ with the type τ. The JavaScript engine handles this naturally, so no interference is required. For example, variable `a` in Fig. 3 is assigned `FakedObject` at line 1 by `JSForce`. At line 4, the variable `a` is retyped as a `string` object.

$$\textrm{R-ASSIGN}\quad \frac{\Gamma \vdash_{ths} e_0 : \varphi_{fo} \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash_{ref} e_0 = e_1 : \tau}$$

$$\textrm{R-CALL1}\quad \frac{\tau_0 \trianglerighteq Obj(Function(this : \tau'; \lceil 0 \rceil : \tau_1, ..., \lceil n\text{–}1 \rceil : \tau_n, \varrho \rightarrow \tau))(\varrho') \quad \Gamma \vdash_{ref} e_0 : \varphi_{fo}/\tau}{\vdash_{upd} e_0 : \varphi_{fo}, \varrho' @\tau \mapsfrom \varphi_{ff}, \Gamma \vdash_{ref} e_0(e_1, ..., e_n) : \varphi_{fo}/\perp}$$

$$\textrm{R-CALL2}\quad \frac{\begin{array}{c}\tau_0 \trianglerighteq Obj(Function(this : \tau'; \lceil 0 \rceil : \tau_1, ..., \lceil n\text{–}1 \rceil : \tau_n, \varrho \rightarrow \tau))(\varrho') \\ \Gamma \vdash_{ref} e_0 : \tau_0/\tau' \\ \Gamma \vdash e_1 : \tau_1 \qquad ...\Gamma \vdash e_{(i-1)} : \tau_{(i-1)} \qquad \Gamma \vdash e_i : \varphi_{fo} \qquad \Gamma \vdash e_{(i+1)} : \tau_{(i+1)} \qquad ... \qquad \Gamma \vdash e_n : \tau_n\end{array}}{\vdash_{upd} e_i : \varphi_{fo} @\tau \mapsfrom \tau_i, \Gamma \vdash_{ref} e_0(e_1, ..., e_n) : \tau/\perp}$$

$$\textrm{R-NEW}\quad \frac{\tau_0 \trianglerighteq Obj(Function(this : \tau'; \lceil 0 \rceil : \tau_1, ..., \lceil n\text{–}1 \rceil : \tau_n, \varrho \rightarrow \tau))(\varrho') \quad \Gamma \vdash_{ref} e_0 : \varphi_{fo}/\tau}{\vdash_{upd} e_0 : \varphi_{fo}, \varrho' @\tau \mapsfrom \varphi_{ff}, \Gamma \vdash_{ref} new\ e_0(e_1, ..., e_n) : \varphi_{fo}/\perp}$$

$$\textrm{R-BINOPERATOR1}\quad \frac{\Gamma \vdash e_1 : \varphi_{fo} \qquad \Gamma \vdash e_2 : \tau' \qquad \neg(e_2\ is\ \varphi_{fo})}{\vdash_{upd} e_1 : \varphi_{fo} @\tau \mapsfrom \tau', \Gamma \vdash e_1\ op\ e_2 : \tau'}$$

$$\textrm{R-BINOPERATOR2}\quad \frac{\Gamma \vdash e_1 : \varphi_{fo} \qquad \Gamma \vdash e_2 : \varphi_{fo}}{\vdash_{upd} e_1 : \varphi_{fo} @\tau \mapsfrom \varphi_n, \vdash_{upd} e_2 : \varphi_{fo} @\tau \mapsfrom \varphi_n, \Gamma \vdash e_1\ op\ e_2 : \tau}$$

$$\textrm{R-INDEX1}\quad \frac{\Gamma \vdash e_1 : \varphi_{fo} \qquad \tau_1 \trianglerighteq Obj(\varphi_1)(\varrho_1) \qquad \Gamma \vdash e_2 : \varphi_n}{\vdash_{upd} e_1 : \varphi_{fo} @\tau \mapsfrom \tau_1, \Gamma \vdash_{ths} e_1[e_2] : \varphi_{fo}}$$

$$\textrm{R\_UNARYOPERATOR}\quad \frac{\Gamma \vdash e_1 : \varphi_{fo}}{\vdash_{upd} e_2 : \varphi_{fo} @\tau \mapsfrom \varphi_n, \Gamma \vdash op\ e_1 : \tau}$$

$$\textrm{R-INDEX2}\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 \trianglerighteq Obj(\varphi_1)(\varrho_1) \qquad \Gamma \vdash e_2 : \varphi_{fo} \qquad \vdash_{upd} \varrho_1 @\varphi_n \mapsto \tau'}{\vdash_{upd} e_2 : \varphi_{fo} @\tau \mapsfrom \varphi_n, \Gamma \vdash_{ths} e_1[e_2] : \tau'}$$

Fig. 5. Typing rules

(2) *R-CALL1* and *R-NEW*. These two rules describe the typing rule for the scenario when a `FakedObject:FObj` is used as a function call or by the `new` expression. Function calls and the `new` expression both expect their first operand to evaluate to a function. So, `JSForce` updates the `FakedObject:FObj` to `FakedFunction:FFun` for this situation. The `FakedFunction` is a special function object which is configured to accept arbitrary parameters. The return value of the function is set to a `FakedObject:FObj` so that it can be retyped whenever necessary.

(3) *R-CALL2*. This rule describes the case where the callee is a known function, but a `FakedObject:FObj` is passed as a function parameter. `JSForce` types the `FakedObject:FObj` to the required type of the callee's arguments. The JavaScript language has many standard built-in libraries such as `Math` and `Date`. When a `FakedObject:FObj` is used by the standard library function, we update the type based upon the specification of the library function [1]. Currently, `JSForce` implements retyping for several common libraries (e.g., `Math`, `Number`, `Date`).

(4) *R-BINOPERATOR1/2* and *R-UNARYOPERATOR*. These three rules describe how to update the type if the `FakedObject:FObj` is involved in an

expression with an operator. `JSForce` updates the `FakedObject:FObj`'s type based upon the semantics of the operator. For unary operators, it is straightforward to determine the type from the operator's semantics. For instance, the postfix operator indicates the type as `number`. For binary operators, the typing becomes more complicated. If both operands are `FakedObject:FObj` and the operator does not reveal the type of the operands, `JSForce` types them to `number`. This is because the `number` type can be converted to most types naturally by the JavaScript engine. For example, the `number` type in JavaScript can be converted to the `string` type, but it may fail to convert a `string` to a `number`. Later during execution, if the types can be determined, `JSForce` will update the type to the correct type. If only one of the two operands is `FakedObject:FObj`, `JSForce` determines the type based upon the other operand's type and the operator's semantics.

(5) *R-INDEX1* and *R-INDEX2*. These two rules describe how to update the type when there are indexing operations. A `FakedObject:FObj` is updated to an $ArrayObject : \phi_o$ whenever a key is used as an array index to access elements of the `FakedObject`. `JSForce` creates an *ArrayObject* and initializes the elements to `FakedObject:FObj`. The length of the *ArrayObject* is set to 2**CurrentIndex*. If an Out-Of-Boundary access is found, `JSForce` doubles the length of *ArrayObject*. If the array index is `FakedObject`, `JSForce` types it to `number` and initializes it as `0`, which avoids Out-Of-Boundary exceptions. If both the index object and base object are `FakedObject:FObj`, the R-INDEX2 rule is first applied to update the index object to `number`, then the R-INDEX1 rule is applied to update the base object to *ArrayObject*.

Example. Figure 4 presents a forced execution of the sample shown in Fig. 3. In the execution, the branch in lines 8–11 is not taken. At line 1, `JSForce` assigns a `FakedObject:Fobj` to `a`, instead of `null`. This is because at line 3 the access to property `length` raises an exception if `a` is `null`. At line 2, we can see a `FakedObject:FObj` is first assigned to `c`. Once `c` is added to `1`, `JSForce` updates the value of `c` to a random number. Lines 6 and 7 show that if a `FakedObject:FObj` is used in the function call or `new` expression, `JSForce` updates it to `FakedFunction:FFun`. The return value of the faked function is still configured to `FakedObject:FObj`, so that at line 13, `d` is updated to hold a random number.

`JSForce` also automatically recovers from other exceptions by intercepting those exceptions to eliminate the exception condition. For example, `JSForce` will update a divisor to a non-zero value if a division-by-zero exception is raised.

3.2 Path Exploration in `JSForce`

One important feature of `JSForce` is the capability of exploring different execution paths of a given JavaScript snippet to expose its behavior and acquire complete analysis results. In this subsection, we explain the path exploration algorithm and strategies.

Algorithm 1. Path Exploration Algorithm

Definitions: *switches* - the set of switched predicates in a forced execution, denoted by a sequence of predicate offsets in the source file(SrcName:offset). For example, $t.js : 15 \cdot t.js : 83 \cdot t.js : 100$ means the branch in source file $t.js$ with the offset 15, 83, 100 is switched. EX, WL - a set of forced executions, each denoted by a sequence of switched predicates. $preds : \overline{Predicate \times boolean}$ - the sequence of executed predicates.

```
Input: The tested $JS$
Output: $FULL\_EX$
1: $FULL\_EX \leftarrow \emptyset$
2: $SRC \leftarrow \{JS\}$
3: while $SRC$ do
4:     $WL \leftarrow \{\emptyset\}$
5:     $EX \leftarrow \emptyset$
6:     $js \leftarrow SRC.pop()$
7:     while $WL$ do
8:         $switches \leftarrow WL.pop()$
9:         $EX \leftarrow EX \cup switches$
10:        $(preds, newJS) \leftarrow$ ExecuteCode$(js, switches)$
11:        $SRC \leftarrow SRC \cup newJS$
12:        $t \leftarrow len(switches)$
13:        $preds \leftarrow remove\ the\ first\ t\ elements\ in\ preds$
14:        for all $(p, b) \in preds$ do
15:            if $!covered(p, \neg b)$ then
16:                $WL \leftarrow WL \cup switches \cdot (p, b)$
17:            end if
18:        end for
19:    end while
20:    $FULL\_EX \leftarrow FULL\_EX \cup \{EX : js\}$
21: end while
22: procedure ExecuteCode($JS, switches$)
23:    $preds \leftarrow switches$
24:    $CBQ \leftarrow \emptyset$
25:    $newJS \leftarrow \emptyset$
26:    for all $stmt \in JS$ do
27:        if $isNoneEvalFunctionCallStmt(stmt)$ then
28:            if $CalleeTakesStrings(stmt)$ then
29:                $newJS \leftarrow newJS \cup GetJSFromString(stmt)$
30:            end if
31:            if $CalleeRegisterCallback(stmt)$ then
32:                $CBQ \leftarrow CBQ \cup ExtractCBFunc(stmt)$
33:            end if
34:        else if $isBranchStmt(stmt)$ then
35:            if $GetSwitch(stmt) \in switches$ then
36:                $Execute\ according\ to\ switches$
37:            else
38:                $preds \leftarrow preds \cdot GetPredicate(stmt)$
39:            end if
40:        end if
41:    end for
42:    for all $cb \in CBQ$ do
43:        $(preds', newJS') \leftarrow$ ExecuteCode$(cb, \emptyset)$
44:        $newJS \leftarrow newJS \cup newJS'$
45:        $preds \leftarrow preds \cdot preds'$
46:    end for
       return $(preds, newJS)$
47: end procedure
```

In practice, attackers constantly adopt the dynamic features of JavaScript to evade detection. This results in incomplete path exploration under two circumstances. The first is when strings are dynamically generated. For instance, `document.write` is often abused to inject dynamically decoded malicious JavaScript code into the page at runtime. The second is when event callbacks are used. As discussed in Sect. 2, attackers can abuse event callbacks to stop the execution of malicious code. `JSForce` solves this by employing specific path exploration strategies. Within the execution, if faked functions take strings as input, `JSForce` examines the strings and executes the code if they contain JavaScript. This strategy is only applied on faked functions since original functions (`eval`) can handle the strings as defined. `JSForce` also detects the callback registration function and invokes the callback function immediately after the current execution terminates.

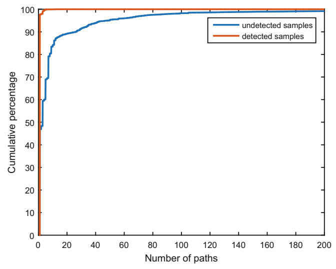
`JSForce` treats `try-catch` statements as `if-else` statements, ie., it executes each `try` block and `catch` block separately. Ternary operators are also treated as `if-else` statements: both values are evaluated.

There are several different path exploration algorithms: linear search, quadratic search, and exponential search [17]. The goal of path exploration in `JSForce` is to maximize the code coverage to improve the detection rate of mali-

Table 1. Effectiveness results.

Sample set	Total	Without JSForce	With JSForce	Improvement
Old HTML	66,325	193	357	84.9%
New HTML	106,018	2,250	20,649	817.3%
HTML total	**172,995**	**2,443**	**21,006**	**759.8%**
Old PDF	22,081	6,306	6,475	2.7%
New PDF	1,428	32	170	431.2%
PDF total	**23,509**	**6,338**	**6,645**	**4.8%**

Table 2. Num of path exploration during analysis.



cious payload with an acceptable performance overhead. Quadratic and exponential searches are too expensive, so `JSForce` employs the linear search only.

Algorithm 1 describes the path exploration algorithm, which generates a pool of forced executions that achieve maximized code coverage. The complexity is $O(n)$, where n is the number of JavaScript statements. n may change at runtime because JavaScript code can be dynamically generated. Initially, `JSForce` executes the program without switching any predicates since `switches` is initialized as $\emptyset$ (line 8) for the first time. `JSForce` executes the program according to the `switches` at line 10 and returns `preds` and dynamically generated code `newJS`. In lines 12–17, we determine if it would be of interest to further switch more predicate instances. Lines 11–13 compute the sequence of predicate instances eligible for switching. Note that it cannot be a predicate before the last switched predicate specified in `switches`. Switching such a predicate may change the control flow such that the specification in `switches` becomes invalid. Specifically, line 16 switches the predicate if the other branch has not been covered. In each new forced execution, we essentially switch one more predicate.

The procedure `ExecuteCode` (lines 22–47) describes the execution process. It collects dynamically generated JavaScript code (lines 28–30) and the executed predicates (lines 34–38). The new generated JavaScript code, `newJS`, will be executed after the path exploration of the current `js` finishes. The registered callback functions (lines 31–33) are also queued and invoked after the current execution finishes (lines 42–46). As an example, recall the callback function `redir()` used in line 16 of Fig. 1. Instead of waiting for the timeout, `JSForce` will trigger the `redir()` function immediately after the current execution finishes.

4 Evaluation

`JSForce` is implemented by extending the V8 JavaScript engine [5] on the X86-64 platform. It is comprised of approximately 4,600 lines of C/C++ and 1,500 lines of Python. In this section, we present details on the evaluation of effectiveness and runtime performance of `JSForce` using a large number of real-world samples.

4.1 Dataset and Experiment Setup

Dataset. The dataset used for our evaluation consists of two sets: a malicious sample set and a benign sample set. For the malicious set, we collected a sample set with 172,995 HTML files and 23,509 PDF files from various databases. For the benign sample set, we crawled the Alexa top 100 websites [2] and collected 47,592 HTML files.

Experiment Setup. For JavaScript code analysis, we leverage the jsunpack [11] tool. Jsunpack is a widely used malicious JavaScript code analysis tool that utilizes the SpiderMonkey JavaScript engine for code execution. For the sake of our evaluation, we replaced the SpiderMonkey from jsunpack with `JSForce` and relied upon the detection policies in jsunpack for malicious code detection. We conducted our experiments on a test machine equipped with Intel(R) Xeon(R) E5-2650 CPU (20M Cache, 2 GHz) and 128 GB of physical memory. The operating system was Ubuntu 12.04.3 (64 bit).

4.2 Effectiveness

For the evaluation of effectiveness, we would like to demonstrate that `JSForce` can indeed help the malicious JavaScript code analysis by performing efficient forced execution. In order to achieve that, we utilize our malicious HTML and PDF sample sets and run the sample sets against jsunpack both with or without `JSForce` for the evaluation. In the interest of showing how useful our faked object retyping is, we also conduct another experiment that disables the retyping and only keeps the reference error recovery component and path exploration component.

Experimental Results. Table 1 illustrates the experimental results for effectiveness. It demonstrates that `JSForce` could greatly improve the detection rate for JavaScript analysis. We can see detection rate improvements of 759.84% and 4.84% for HTML and PDF samples, respectively, when using `JSForce`-extended jsunpack instead of the original version for analysis. And all the samples detected by original jsunpack are also flagged by `JSForce`-extended jsunpack. We further break down the numbers into old and new sample sets and perceive that the extended version could perform much better than original jsunpack in analyzing new samples. For new HTML samples, jsunpack with `JSForce` is able to detect 817.3% more samples while for old samples, the number is 84.97%. Similar results are also observed for PDF samples. After manual inspection, we confirmed that this is because many of the old samples have been analyzed for quite sometime and jsunpack already has the signatures stored in its database, leaving only a small margin for `JSForce` to improve upon. For the faked object retyping evaluation, we reran the test using 106,018 new HTML malicious samples with retyping component disabled. The result shows that only 8,677 samples can be detected by `JSForce` in contrast to 20,649 with retyping enabled. This result reveals the usefulness of our faked object retyping component during analysis. Nevertheless, through our experiments, we are able to draw the conclusion that `JSForce` is quite effective for boosting the effectiveness of JavaScript analysis.

Number of Paths Explored. Potentially, there may be a large number of paths that exist inside of a single JavaScript program. The effectiveness and efficiency of `JSForce` are closely related to the number of paths explored during analysis. Hence, we would like to show some statistics on the number of paths that `JSForce` explored during analysis.

The result depicted in Table 2 shows that `JSForce` is able to detect the maliciousness of samples with a limited number of path explorations. An interesting observation is that over 96% of the samples were detected by exploring only a single path. Even though most of the analysis for detected samples can be finished by exploring just one path, the path exploration of `JSForce` is still essential. Note that 98% of the samples missed by the default jsunpack, but detected by the `JSForce`-extended version, explore at least two paths. So, the analysis could still receive an enormous benefit from `JSForce` in terms of path exploration. As for any undetected samples, `JSForce` will explore the entire code space during analysis, which requires a larger amount of path exploration and longer analysis runtime.

4.3 Runtime Performance

In this section, we evaluate the runtime performance of `JSForce` by using our malicious and benign datasets with a comparison between the original jsunpack and the `JSForce`-extended version.

Runtime for Detected Samples. In this section, we compare the runtime performance using the HTML and PDF samples that can be detected by jsunpack both with and without `JSForce`. The reason why we chose this sample set is that we wished to observe whether the `JSForce`-extended version can achieve efficiency comparable to the original jsunpack when using a detectable malicious sample. The results are displayed in Figs. 6 and 7. The results conclude that `JSForce`-extended version has better runtime performance than jsunpack for over 90.9% of HTML and 83.6% of PDF samples. This conclusion is quite surprising as the `JSForce`-extended version tends to explore multiple paths while jsunpack only probes for one.

In theory, jsunpack should have better runtime performance. However, after investigation, we found that many of the JavaScript samples require specific system configurations (such as specific browser kernel version) to run. As a result, when jsunpack performs analysis, it will run the JavaScript programs under multiple settings. This results in multiple executions, which take additional time to complete. In contrast, the `JSForce`-extended version handled this issue with forced execution, resulting in better runtime performance in practice.

Runtime for Undetected Samples. Figures 8 and 9 show the runtime performance of `JSForce` for undetected samples. We empirically set the time limit to be 300

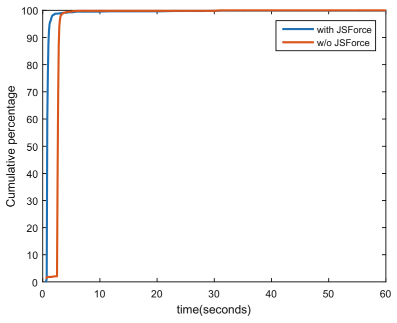


Fig. 6. Runtime for detected HTML.

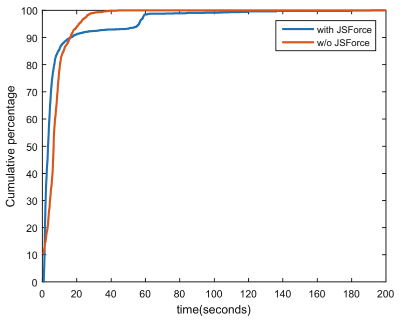


Fig. 7. Runtime for detected PDF.

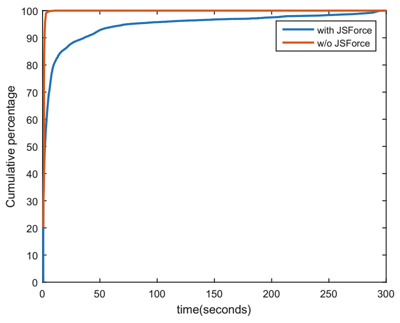


Fig. 8. Runtime for undetected HTML.

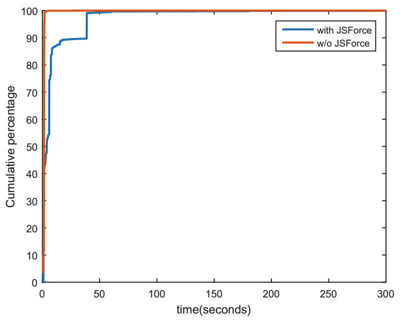


Fig. 9. Runtime for undetected PDF.

s in consequence of the fact that experiment shows almost all (99.6%) HTML and PDF samples can be analyzed within 300 s. As demonstrated in the figures, the average analysis runtime for HTML and PDF samples are 12.02 and 8.15 s, while the analysis for a majority (80%) of HTML samples and PDF samples are finished within 8.54 and 7.4 s, respectively. When compared with the original jsunpack, the `JSForce`-extended version achieves an average runtime of 16.08 s and 7.97 s for undetected HTML and PDF samples while jsunpack finishes execution in 1.13 s and 1.37 s, correspondingly. Our conclusion from these experiments are that the performance overhead of `JSForce` is quite reasonable and can certainly meet the requirements of large scale JavaScript analysis.

5 Conclusion

In this paper, we presented the design and implementation of a novel JavaScript forced execution engine named `JSForce` which enables non crashable execution model while ensuring complete code coverage. We evaluated `JSForce` using a large number of HTML and PDF samples. Experimental results showed that by adopting `JSForce`, the malicious JavaScript detection rate can be greatly improved without any noticeable false positive increase and the runtime overhead was generally neglectable.

Acknowledgments. We would like to thank anonymous reviewers for their feedback. This research was supported in part by National Science Foundation Grant #1664315, and DARPA Grant #FA8750-16-C-0044. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

1. http://www.ecmascript.org/
2. http://www.alexa.com/topsites
3. 2015 symantec internet security threat report. https://goo.gl/UIPdR8
4. Sputnik. https://code.google.com/p/sputniktests/
5. V8 javascript engine. https://developers.google.com/v8/
6. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. In: Lee, W., Wang, C., Dagon, D. (eds.) Botnet Detection. ADIS, vol. 36, pp. 65–88. Springer, Heidelberg (2008). https://doi.org/10.1007/978-0-387-68768-1_4
7. Cao, Y., Pan, X., Chen, Y., Zhuge, J.: JShield: towards real-time and vulnerability-based detection of polluted drive-by download attacks. In: Proceedings of Annual Computer Security Applications Conference (ACSAC) (2014)
8. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious javascript code. In: Proceedings of the 19th International Conference on World Wide Web (2010)
9. Curtsinger, C., Livshits, B., Zorn, B.G., Seifert, C.: Fast and precise in-browser javascript malware detection. In: USENIX Security Symposium, Zozzle (2011)
10. Feinstein, B., Peck, D., SecureWorks, I.: Caffeine monkey: automated collection, detection and analysis of malicious javascript. In: Black Hat USA (2007)
11. Hartstein, B.: Jsunpack: an automatic javascript unpacker. In: ShmooCon Convention (2009)
12. Kapravelos, A., Shoshitaishvili, Y., Cova, M., Kruegel, C., Vigna, G.: Revolver: an automated approach to the detection of evasive web-based malware. In: USENIX Security, pp. 637–652. Citeseer (2013)
13. Kolbitsch, C., Livshits, B., Zorn, B., Seifert, C.: Rozzle: de-cloaking internet malware. In: 2012 IEEE Symposium on Security and Privacy (SP) (2012)
14. Lu, G., Debray, S.: Automatic simplification of obfuscated javascript code: a semantics-based approach. In: Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability (2012)
15. Mowery, K., Bogenreif, D., Yilek, S., Shacham, H.: Fingerprinting information in javascript implementations. In: Proceedings of W2SP, vol. 2 (2011)
16. Mulazzani, M., Reschl, P., Huber, M., Leithner, M., Schrittwieser, S., Weippl, E., Wien, F.: Fast and reliable browser identification with javascript engine fingerprinting. In: Web 2.0 Workshop on Security and Privacy (W2SP), vol. 5 (2013)
17. Peng, F., Deng, Z., Zhang, X., Xu, D., Lin, Z., Su, Z.: X-force: force-executing binary programs for security applications. In: Proceedings of the 2014 USENIX Security Symposium, San Diego, CA, August 2014 (2014)
18. Ratanaworabhan, P., Livshits, B., Zorn, B.: Nozzle: a defense against heap-spraying code injection attacks. In: Proceedings of the USENIX Security Symposium (2009)
19. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: 2010 IEEE Symposium on Security and Privacy (SP) (2010)

20. 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). https://doi.org/10.1007/978-3-540-31987-0_28
21. Trinh, M.-T., Chu, D.-H., Jaffar, J.: S3: a symbolic string solver for vulnerability detection in web applications. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1232–1243. ACM (2014)
22. Upathilake, R., Li, Y., Matrawy, A.: A classification of web browser fingerprinting techniques. In: 2015 7th International Conference on New Technologies, Mobility and Security (NTMS). IEEE (2015)
23. Wang, D.Y., Savage, S., Voelker, G.M.: Cloak and dagger: dynamics of web search cloaking. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 477–490. ACM (2011)
24. Xu, W., Zhang, F., Zhu, S.: The power of obfuscation techniques in malicious javascript code: a measurement study. In: 2012 7th International Conference on Malicious and Unwanted Software (MALWARE), pp. 9–16. IEEE (2012)