10-2017

JSForce: A forced execution engine for malicious javascript detection

Xunchao HU

Yao CHENG

Yue DUAN
Singapore Management University, yueduan@smu.edu.sg

Andrew HENDERSON

Heng YIN

Citation

HU, Xunchao; CHENG, Yao; DUAN, Yue; HENDERSON, Andrew; and YIN, Heng. JSForce: A forced execution engine for malicious javascript detection. (2017). *Proceedings of the 13th EAI International Conference on Security and Privacy in Communication Networks, Ontario, Canada, 2017 October 22-25*. 238, 704-720.
Available at: https://ink.library.smu.edu.sg/sis_research/8172

JSForce: A Forced Execution Engine for Malicious JavaScript Detection

Xunchao Hu, Yao Cheng, Andrew Henderson
Department of EECS,
Syracuse University, USA
Email: {xhu31, ycheng, anhender}@syr.edu

Yue Duan, Heng Yin
Department of CSE,
University of California, Riverside
Email: {yduan005, heng}@cs.ucr.edu

***Abstract*—The drastic increase of JavaScript exploitation attacks has led to a strong interest in developing techniques to enable malicious JavaScript analysis. 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. We also make `JSForce` publicly available as an online service and will release the source code to the security community upon the acceptance for publication.**

I. Introduction

Malicious JavaScript has become an important attack vector for software exploitation attacks. Attacks in browsers, as well as PDF files containing malicious embedded JavaScript, are typical examples of how attackers launch attacks using JavaScript. According to a recent report from Symantec [8], there are millions of victims attacked by malicious JavaScript on the Internet each day.

In recent years, a number of techniques [17], [29], [22], [21], [26], [35], [18], [25], [16] have been proposed to detect malicious JavaScript code. Due to the dynamic features of the JavaScript language, static analysis [20], [27], [38], [18] can be easily evaded using obfuscation techniques [46]. Consequently, researchers rely upon dynamic analysis [17], [29], [22] to expose the typical features of malicious JavaScript. More specifically, these approaches rely upon visiting websites or opening PDF files with a full-fledged or emulated browser/PDF reader and then monitoring the different features (*eval* strings [22], heap health [35], etc.) 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 [42] 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, Rozzle [26] explores multiple environment related paths within a single execution. 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 recent fingerprinting techniques [31], [32]. These fingerprinting techniques do not rely upon any specific APIs, and thus Rozzle can be evaded because the predefined profile cannot include those fingerprinting techniques. Also, Rozzle may introduce runtime errors because it executes infeasible paths which may stop the analysis before the malicious code is executed. Revolver [25] employs a machine learning-based detection algorithm to identify evasive JavaScript malware. However, it is dependent upon a known sample set and is unable to detect 0-day JavaScript malware. Although symbolic execution of JavaScript [37] can be applied to explore all of the possible execution paths, the performance overhead of a symbolic string solver [41] and the dynamic features of JavaScript make it infeasible for practical use.

In this paper, 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 Section V, 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 [33], iRiS [19]), 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 [11] and evaluate the correctness, 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 forced execution of JavaScript, a 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 correctness, effectiveness, and runtime performance using 220,587 HTML and 23,509 PDF real-world samples. 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.
4) We create an online service and make JSForce publicly available at [9]. Upon the acceptance for publication, we will release the source code of JSForce to the security community.

II. Background and Overview

To provide the reader with a better understanding of the motivation for our system and the problems that it addresses, we begin with a discussion of the malicious JavaScript code used in drive-by-download attacks.

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 that software uses, and then launch one or more exploits.

```
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);
```

Figure 1: The Malicious JavaScript Sample

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 successfully 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).

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 [15] 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 [17], [29], [22]

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 [22], heap health [35], 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 a technique called *cloaking* [43], 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 [26] attempts to address this code coverage problem by exploring environment-related paths within a single execution. For instance, because `att` in Figure 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 [31], [32], [42]. These new techniques do not rely upon any specific APIs. For instance, the JavaScript engine fingerprinting technique [32] relies upon JavaScript conformance tests such as the Sputnik [10] 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 Section III-A. By overcoming those limitations of Rozzle, `JSForce` achieves greater code coverage.

Revolver [25] 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 [15]. This technique, while improving code coverage over dynamic analysis, suffers from scalability challenges and is, in many ways, unnecessarily precise [26]. Within the context of JavaScript analysis, symbolic execution becomes more challenging [37]. 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 [41] 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 [33]), 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 Figure 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*.

III. 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.

A. Forced Execution Semantics

$\langle$*EXPRESSIONS*$\rangle$ ::= c	CONSTANT
$\mid$ x	VARIABLE
$\mid$ $x.f$	FIELD ACCESS
$\mid$ $x.prot$	PROTO ACCESS
$\mid$ e *op* e	BINARY OP
$\mid$ $this$	THIS
$\mid$ $\{f_1 : e_1, ..., f_n : e_n\}$	OBJECT LITERAL
$\mid$ {function($p_1, ..., p_n$){S}}	FUNCTION DEF
$\mid$ f($a_1, ..., a_n$)	FUNCTION CALL
$\mid$ new f($a_1, ..., a_n$)	NEW
$\langle$*STATEMENTS*$\rangle$ ::= skip	SKIP
$\mid$ $S_1 : S_2$	SEQ
$\mid$ $varx$	VAR DECL
$\mid$ $x := e$	ASSIGN
$\mid$ $x.f := e$	ASSIGN
$\mid$ if e then S_1 else S_2	CONDITIONAL
$\mid$ while e do S	WHILE
$\mid$ try{S}catch{S}finally{S}	TRY CATCH
$\mid$ return e	RETURN

Figure 2: Core JavaScript

The JavaScript Language: JavaScript is a high-level, dynamic, untyped, and interpreted programming language. Figure 2 summarizes the syntax of the core JavaScript, which captures the essence of JavaScript. At runtime, the JavaScript engine dynamically interprets JavaScript 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 Figure 1 raises a `ReferenceError` exception because `btt` is not defined. To tolerate such invalid object accesses, forced execution must handle such failures.

Types:		
τ	::=	$\sum_{i \in \mathcal{I}, \mathcal{T} \subseteq \{\bot,u,b,s,n,o\}} \varphi_i$
Rows:		
ϱ	::=	$str{:}\tau, \varrho$
	$\mid$	ϱ_τ
Type environments:		
Γ	::=	$\Gamma(x : \tau)$
	$\mid$	$\varnothing$
Type summands and indices:		
$\varphi_\bot$	::=	Undef
φ_u	::=	Null
φ_b	::=	Bool(ξ_b)
ξ_b	::=	$false \mid true \mid \top$
φ_s	::=	String(ξ_s)
ξ_s	::=	$str \mid \top$
φ_n	::=	Number(ξ_n)
ξ_n	::=	$num \mid \top$
φ_f	::=	Function($this : \tau; \varrho \rightarrow \tau$)
φ_o	::=	Obj($\sum_{i \in \mathcal{I}, \mathcal{T} \subseteq \{b,s,n,f,\bot\}} \varphi_i$)($\varrho$)
φ_{fo}	::=	FObj
φ_{ff}	::=	FFun

Figure 3: Syntax of JavaScript Types

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:

```
1 c = a/2;
2 for(i= c;i<10000;i++)
3 {
4   memory[i] = nop + nop + shellcode;
```

```
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;
```

Figure 4: JavaScript Sample

```
5  }
```

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 [40] is extended to support these two new types. Figure 3 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 raising 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.

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 FakedObject. Line 2 in Figure 4 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 Figure 4 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 Figure 4 is assigned FakedObject at line 1 by JSForce. At line 4, the variable a is retyped as a string object.
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

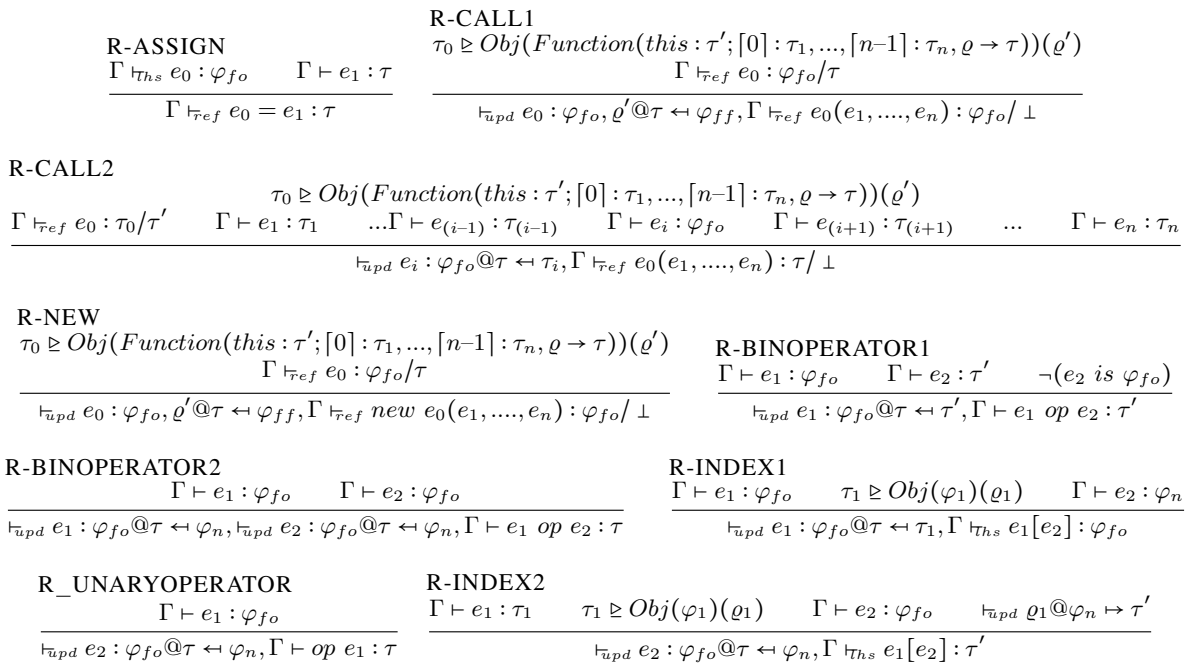

Figure 5: Typing rules

`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: Table I presents a forced execution of the sample shown in Figure 4. 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`. On 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

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

Table I: Forced execution of sample in Figure 4

`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.

B. Path Exploration in `JSForce`

One important functionality 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.

In practice, attackers constantly adopt the dynamic features of JavaScript to aid in evading 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 Section II, 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 [33].

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

The goal of path exploration in `JSForce` is to maximize the code coverage to improve the detection rate of mali-

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 $\varnothing$ (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 Figure 1. Instead of waiting for the timeout, JSForce will trigger the redir() function immediately after the current execution finishes.

IV. Implementation

JSForce is implemented by extending the V8 JavaScript engine [11] on the X86-64 platform. It is comprised of approximately 4,600 lines of C/C++ code and 1,500 lines of Python code. We address some prominent challenges of its implementation in this section.

Reference Error Recovery & Faked Object Retyping: In V8, an abstract syntax tree (AST) is generated for every function, which is then compiled into native code (known as Just-In-Time code). V8 adopts an inline caching technique [23] to accelerate property accesses. If the property access fails, the execution jumps to the V8 runtime system which handles any inline cache misses. If the runtime system is unable to handle an inline cache miss, either due to reference error or type check error, it raises the corresponding exception and stops the execution.

We modify the inline cache miss handling process to enable reference error recovery and faked object retyping. For reference error recovery, JSForce creates and returns the FakedObject for failed object lookup by changing the V8 property access failure handling functions like Runtime_LoadIC_Miss. For faked object retyping, JSForce inserts additional code into runtime methods like Runtime_BinaryOpIC_Miss that is executed prior to the exception being raised. This additional code follows the rules described in Section III-A to conduct the retyping process if the involved operation contains a FakedObject.

Predicates Flip: We have two approaches available to flip the predicates. The first approach is to flip the predicates within the Just-in-Time code. The Just-in-Time code can be optimized (inline caching, etc.) by V8 in accordance with the execution profile. To enable predicates flipping, a runtime function must be inserted before every branch so that JSForce can manipulate the predicate value. This approach may affect the optimization process of Just-in-Time code.

JSForce takes the second approach: if the branch A of a predicate needs to be taken, JSForce replaces the other branch with this branch A. At runtime, no matter which branch is taken, the branch A is executed. For instance, we want to take the $\{A\}$ branch of the statement if(e)$\{A\}$else$\{B\}$. JSForce changes it to if(e)$\{A\}$else$\{A\}$, so that $\{A\}$ is executed at runtime.

Loops and Recursions: Sometimes, JSForce may cause a loop to execute for a very long time, due to the introduction of faked objects. To solve this problem, JSForce inserts a time counter for every loop statement (for...in and for...of are excluded, as they will always terminate), and it will terminate the loop if the execution time exceeds a limit. Similarly, if JSForce forces a predicate that guards the termination of a recursive function call, a very deep recursion may result. To address deep recursion, JSForce monitors the stack depth. Once the maximum call stack size (defined by V8) is reached, calls to that function are omitted by JSForce.

V. Evaluation

In this section, we present details on the evaluation of correctness, effectiveness and runtime performance of JSForce using a large number of real-world samples.

A. Dataset & Experiment Setup

Dataset: The complete dataset used for our evaluation consists of two sample 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 including VirusTotal [12], Contagio [2], MalTrafficAnalysis [3], and Threatglass [4]. Among those, all samples from VirusTotal were new samples evaluated within a month of being submitted, with the samples provided from other sources being relatively old. For the benign sample set, we crawled the Alexa top 100 websites [5] and collected 47,592 HTML files.

Category	Total	Detected by JSForce	Percentage
True Positive	389	389	100%
False Positive	47,592	9	0.019%

Table II: Correctness Results.

Experiment Setup: For JavaScript code analysis, we leverage the jsunpack [22] tool. Jsunpack is a widely used malicious JavaScript code analysis tool that utilizes the SpiderMonkey [6] JavaScript engine for code execution. Six distinct configurations are predefined within jsunpack to maximize the exploration of JavaScript code by trying different browsers and language settings. 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. Most of our experiments are based upon the comparison between the original jsunpack and the JSForce-extended jsunpack. Note that the experiments performed within this paper are only intended to show the *improvement of detection results* over the original ones when adopting JSForce. The detection policy itself is another important research topic which is orthogonal to the focus of this paper. We conducted our experiments on a test machine equipped with Intel(R) Xeon(R) E5-2650 CPU (20M Cache, 2GHz) and 128GB of physical memory. The operating system was Ubuntu 12.04.3 (64bit).

B. Correctness

In this section, we evaluate the correctness of the analysis result for JSForce. The goals of this evaluation are twofold. First, we wish to know the true positive rate of our analysis results, meaning that we wish to verify whether a JavaScript program is undoubtedly malicious if it is tagged as one by the analysis tools. Second, we wish to understand any false positives in the results so as to determine whether any benign JavaScript code can be mistakenly labeled as malicious.

True positive: With our first goal in mind, we queried VirusTotal [12] for malicious HTML files and collected 389 samples which are precisely labeled with specific CVE (Common Vulnerabilities and Exposures) numbers that match CVEs listed in jsunpack. Furthermore, we manually reviewed each of the samples and confirmed the existence of shellcode or malicious signatures. This step is to guarantee all the samples we tested are real malicious samples that should be detected by our tool. Then, we analyzed the samples using jsunpack with JSForce. The experimental result is listed in the first row of Table II as "true positive". It shows that JSForce could successfully detect all of the samples, resulting in a 100% true positive rate. To better understand these results, we further inspected the detailed analysis results to see why our tool tagged samples as malicious. Our inspection results revealed that all of the payload and malicious signatures extracted by the JSForce are indeed malicious, proving that our tool can achieve very high true positive rate with accurate analysis details.

False positive: For our second goal, we analyzed our benign sample set using JSForce and then observed whether any of the samples could be incorrectly labeled as malicious. As shown in the second row of Table II, the JSForce tags 9 out of 47,592 samples as malicious. We first manually confirmed that all 9 samples are clean and thereupon study why the false positives happen. It has been verified by manual inspection that all of the false positives are caused by the inaccurate detection policy, to be more specific, the over-relaxation of the shellcode string matching policy enforced by jsunpack. The reason why our tool could detect them as malicious is that it explores JavaScript code in a more complete fashion in consequence of our forced execution technique. Therefore, based upon the above experimental results, we argue that using JSForce will keep a very low false positive rate for JavaScript code analysis, and is able to assist in accomplishing more thorough results. Theoretically, JSForce can generate higher code coverage than jsunpack and lead to better analysis results. But, the question is by how much. With that, we conducted another set of experiments to show the effectiveness of JSForce.

C. 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 III 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

Sample Set	Total	without JSForce	with JSForce	Improvement	Detected By Both	Missed With JSForce
Old HTML	66,325	193	357	84.9%	193	0
New HTML	106,018	2,250	20,649	817.3%	2250	0
HTML Total	**172,995**	**2,443**	**21,006**	**759.8%**	**2443**	**0**
Old PDF	22,081	6,306	6,475	2.7%	6306	0
New PDF	1,428	32	170	431.2%	32	0
PDF Total	**23,509**	**6,338**	**6,645**	**4.8%**	**6338**	**0**

Table III: Effectiveness Results.

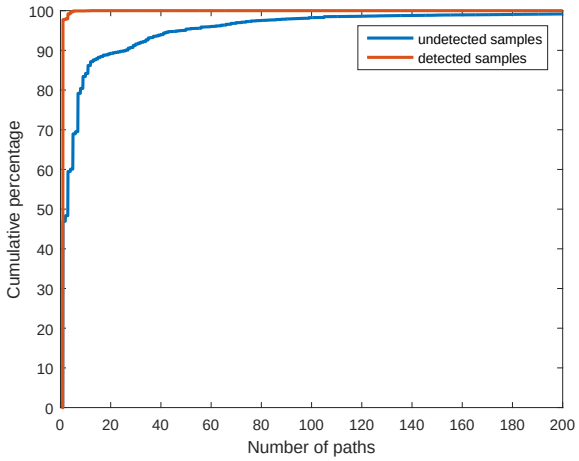

Figure 6: Num of Path Exploration during Analysis.

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 Figure 6 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. Please refer to the Section 6 Case Study for more details on this topic. 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.

D. 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 Figures 7 and 9. 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 10 show the runtime performance of JSForce for undetected samples. We empirically set the time limit to be 300 seconds in consequence of the fact that experiment shows almost all (99.6%) HTML and PDF samples can be analyzed within 300 seconds. As demonstrated in the figures, the average analysis runtime for HTML and PDF samples are 12.02 and 8.15 seconds, while the analysis for a majority (80%) of HTML samples and PDF samples are finished within 8.54 and 7.4 seconds, respectively. When compared with the original jsunpack, the JSForce-extended version achieves

an average runtime of 16.08 seconds and 7.97 seconds for undetected HTML and PDF samples while jsunpack finishes execution in 1.13 seconds and 1.37 seconds, 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.

E. JSForce vs. Rozzle

Ideally, we would like to perform a head-to-head comparison between JSForce and Rozzle using the same dataset. Unfortunately, it is impossible given that neither the Rozzle system nor the dataset used by Rozzle is available for evaluation. It is also nontrivial to implement Rozzle by ourselves. Nevertheless, we can still highlight several advantages of JSForce over Rozzle, from the experimental results reported in that paper.

First, while Rozzle-extended analysis system does, JSForce-extended analysis system does not miss samples detected by the original analysis system. Table IV summarizes the detection results presented in Rozzle paper. Using Rozzle, the experiments extend two malicious JavaScript detection systems - Nozzle [35] and Zozzle [18], and then compare the detection results with the original system using one offline sample set and one online sample set. For the offline experiment, with Rozzle, Nozzle can detect 11,559 samples and gains a significant improvement(11,559 vs. 1,662) over original Nozzle. But it misses 484 (29%) samples which can be detected by original Nozzle. For online experiments, Rozzle-extended configuration also misses 24 (32%) for Nozzle, 225 (8%) for Zozzle respectively. Rozzle paper argues this is because that the runtime errors, introduced when infeasible paths are executed, terminates the execution before the malicious behaviors are exposed. However, since JSForce only collects the path information and no changes are made on the path when the sample is first executed, no runtime errors are introduced by JSForce. Thus as demonstrated in Section V-C, JSForce-extended analysis system can detect all the samples identified by original analysis system while providing the same magnitude improvement as Rozzle's.

Second, JSForce can still function even when the environment setup is incomplete, thanks to the forced execution model (Section III-A), whereas Rozzle may fail due to the runtime errors. This is especially important for low-interaction honey clients like jsunpack. Those low-interaction honey clients emulate the behaviors of browsers or PDF readers, and it is quite challenging to construct a complete environment setup for the tested samples. As discussed in Section VI, of the malicious samples missed by jsunpack, 96.5% are because of the runtime errors caused by incomplete emulation of the running environments for JavaScript code. Since low-interaction honey clients are widely deployed in industry, we argue that JSForce would benefit the industry more than Rozzle.

Third, as discussed in the limitation part of Rozzle paper, Rozzle is less effective for the case that the evasive code triggers the malware execution only when a user interaction occurs, or when a timer fires. We searched the samples missed by jsunpack with the keywords like "onclick" or "settimeout". we found that 80.6% of them deploy timers or user interaction callbacks. JSForce's path exploration algorithm discovers the callback functions during the execution, and invokes them after the current run terminates. However, Rozzle may miss the malicious code hidden in callback functions.

Fourth, Rozzle cannot handle latest fingerprinting techniques discussed in Section II. While we have not found samples deploying these techniques in our dataset, we believe that the attacker will deploy those new fingerprinting techniques with the advancement of anti-evasion techniques in the future. So JSForce is one step ahead of the attacker.

VI. Case Study

```
try{++(document["body"]);
} catch(e) {[shellcode]}
```

[a]

```
window.addEvent('load',
  function() {[shellcode
  ]})
```

[b]

```
function frmAdd() {
  var ifrm = document.createElement('iframe');
  ifrm.style.position='absolute';
  document.body.appendChild(ifrm);};
frmAdd();
```

[c]

```
var se_show_newupdates = new Hash.Cookie('
    se_show_newupdates', {duration: 3600});
```

[d]

```
gapi.load("gapi.iframes", function() {[...]});
```

[e]

Figure 11: Case Study Samples

To better understand the benefits of JSForce, we conducted a case study on 10,975 unique JavaScript code samples missed by jsunpack but detected by JSForce-extended version. The reasons of failed detection using jsunpack can be divided into the following two categories.

Malicious code branch is not triggered: Of those 10,975 samples missed by jsunpack, we found that 10,792 (98.33%) samples are explored by at least two paths when

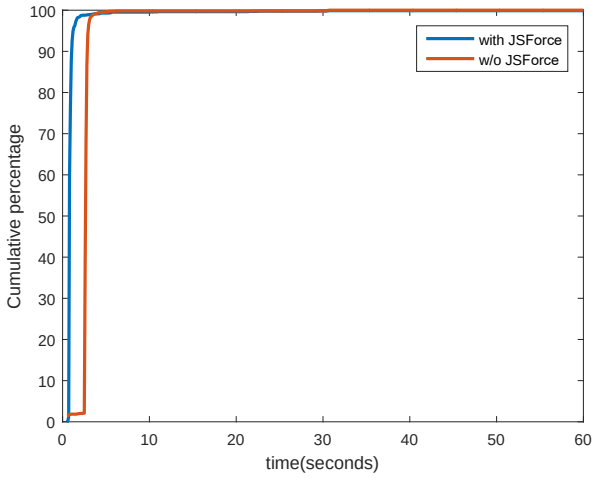

Figure 7: Runtime for Detected HTML samples.

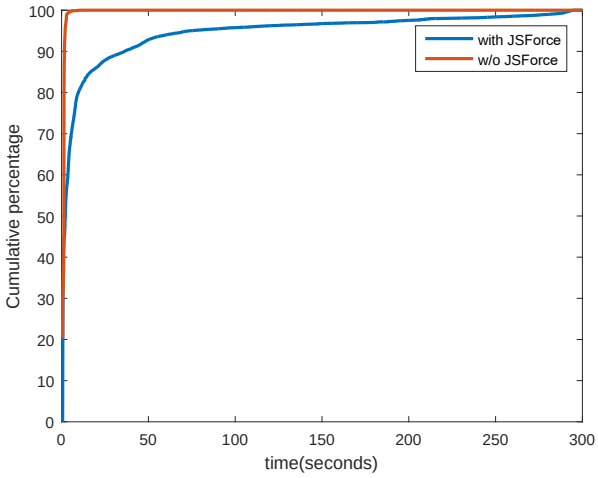

Figure 8: Runtime for Undetected HTML samples.

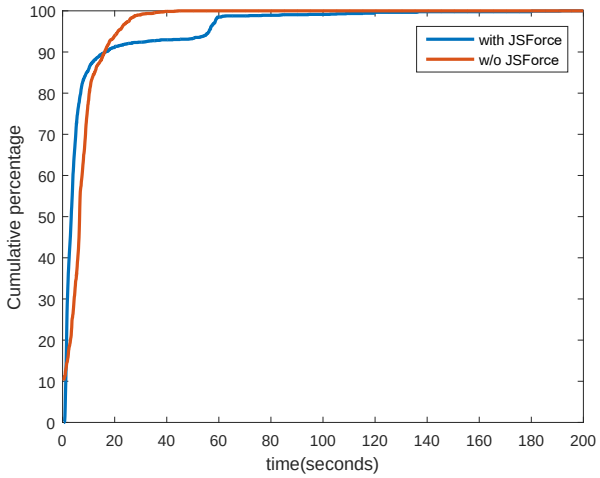

Figure 9: Runtime for Detected PDF samples.

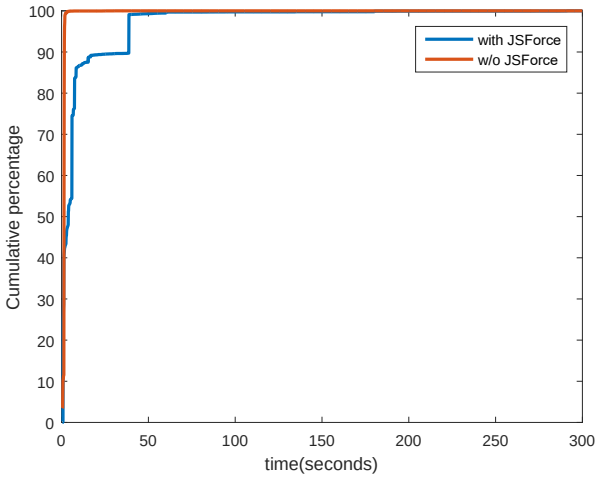

Figure 10: Runtime for Undetected PDF samples.

AnalysisSystem	SampleSet	Conf. w/o Rozzle	Conf. With Rozzle	DetectedByBothConf	MissedByRozzle-extendedConf.
Nozzle	Offline	1,662	11,559	1,178	484(29%)
	Online	74	224	50	24(32%)
Zozzle	Online	2,735	2,660	2,510	225(8%)

Table IV: Detection Results With/Without Rozzle-extended Configuration

using `JSForce`. Although jsunpack attempts to run the sample several times with different configurations to increase the chance of triggering the malicious code branch, it is usually ineffective to do so. This is because it is impossible to emulate every single combination of browser/PDFReader/plugin. In contrast, `JSForce` can explore these paths regardless of the configuration.

The sample in Figure 11(a) hides malicious code within a `catch` block. The attacker attempts to increase the value `document[body]` as a number, which will raise an exception when executed within a real browser. However, it does not raise an exception in jsunpack since its SpiderMonkey engine returns `NaN` for this operation. In fact, the V8 engine used in `JSForce` also exhibits the same behavior as SpiderMonkey. But, the `catch` block is triggered by the path exploration process, so the malicious behavior is still revealed.

Other samples hide code within event callbacks. For instance, the sample in Figure 11(b) registers a callback function using the `window.addEvent` function. jsunpack

fails to invoke the callback function due to the incorrect definition of `window.addEvent` used by jsunpack. At runtime, `JSForce` identifies `window.addEvent` as a callback registration function because an anonymous function is passed to it as the parameter. Then, this anonymous function is queued and invoked at the end of execution.

Execution fails due to runtime errors: Another reason why jsunpack may fail to detect malicious JavaScript code is that the execution can fail due to runtime errors. As we conducted the evaluation, only 230 out of the 10,975 samples could be executed without any runtime errors under the six configurations. Moreover, 10,592 out of 10,975 (96.5%) failed all six configurations, rendering jsunpack completely useless when facing them. These exceptions terminate the execution before the malicious code is executed. The raised exceptions are because of the inaccurate emulation of the running environment for JavaScript code. Examining these exceptions can help security researchers improve jsunpack by supplying more precise emulation environment, which is another benefit that `JSForce` can provide.

One interesting thing about jsunpack is that it tries to fix `ReferrenceError` by providing a definition for this undefined object once `ReferrenceError` is captured. While this fix eliminates the `ReferrenceError`, it often introduces `SyntaxError` or `TypeError` at runtime. `ifrm.style` is not defined in the sample in Figure 11(c). So jsunpack generates code `var ifrm.style = 1` for this sample. Unfortunately, it contains an unexpected token dot. This raises a `SyntaxError` exception. Another way to improve this is to assign `ifrm.style` an `Object` so that `SyntaxError` is avoided and `ifrm.style` can be typed following the typing rules of the JavaScript engine. However, as discussed in Section III-A, this can still cause an exception or lead to unnecessary loss of precision. This case demonstrates the advantage of type inference model deployed by `JSForce`. Although `JSForce` cannot tolerate `SyntaxError`, the type inference model guarantees no further `TypeError` or `SyntaxError` will be introduced.

The sample in Figure 11(d) raises a `TypeError` exception since `Hash.Cookie` is not a constructor. Another sample in Figure 11(e) also raises a `TypeError` exception because `gapi.load` is not a function. `JSForce` can avoid this by applying faked object retyping technique. From another perspective, these two cases manifest the weakness of jsunpack that `Hash.Cookie` and `gapi.load` are not correctly defined. Therefore, as another application, `JSForce` can be used to evaluate the weakness of dynamic JavaScript analysis systems, so security researchers can further improve the systems respectively.

VII. Limitations

If the syntax of the tested JavaScript code is not correct, `JSForce` drops the analysis immediately. The forced execution can introduce syntax error under some cases.

```
var f = function() {
    if (true) {
        function g()
    {return 1;}
    } else {
        function g()
    {return 2;}
      };
    function g(){
    return 3;};
    return g();
    function g(){
    return 4;}
}
```

Figure 12: A JavaScript Sample Interpreted Differently by Different JavaScript Engines

For instance, the parameter of `eval` is supposed to be correct JavaScript code. When the parameter is calculated from faked strings created by `JSForce`, the parameter may become syntax incorrect for `eval`. In the future, we expect to develop techniques [14] to automatically fix the syntax error to enable maximized execution of the code.

While JavaScript language has the official specification from the ECMAScript community [1], the implementation of the language on different JavaScript engines differs slightly because of the complex features and rapid evolving of JavaScript language. The attacker can exploit this weakness to create a deliberate script which exhibits differently on `JSForce` to evade the analysis. Maffeis et al. [30] discussed such an example presented in Figure 12. This code defines a function `f` whose behavior is given by one of the declarations of `g` inside the body of the anonymous function that returns `g`. However, different implementations disagree on which declaration determines the behavior of `f`. Specifically, a call to `f()` should return 4, according to ECMA specification. SpiderMonkey returns 4, while Rhino and Safari return 1, and JScript and the ECMA4 reference implementations return 2. Attackers can leverage these differences to hide the decoding key and evade analysis. To counter this, we can implement `JSForce` on top of different JavaScript engines, such as SpiderMonkey [6] and Chakra [7].

The current path exploration algorithm can efficiently explore most of the sample in a decent time. However, there are still some cases that take a considerable length of time to finish. To exploit this limitation, attackers may place the malicious code deep in the code logic, such that `JSForce` could not reach it within a predefined duration. Note that this limitation is not unique for `JSForce`. All the path exploration techniques share the same limitation. We leave it as future work to develop better path exploration algorithms and search heuristics.

VIII. Related Work

Malicious JavaScript Analysis: In the last few years, there have been a number of approaches to analyzing JavaScript code. They can be roughly divided into two categories-static approach, dynamic approach.

Static Approach. Several systems have focused on statically analyzing JavaScript code to identify malicious web

pages [20], [27], [38], [18]. ZOZZLE [18], in particular, leverages features associated with AST context information (such as, the presence of a variable named shellcode in the context of a loop), for its classification. Since dynamic features of JavaScript plague the static analysis , researchers try to model those features to improve the static analysis result [34], [39], [13].

Dynamic Approach. Dynamic analysis is widely deployed to expose behaviors of obfuscated JavaScript code. Previous work [17], [29], [22] execute JavaScript using an emulated JavaScript running environment and acquire de-obfuscated JavaScript code. To de-obfuscate malicious JavaScript code, Gen et al. [29] simplify the obfuscated JavaScript code by preserving the semantics of the observational equivalence. JSGuard [21] proposed a methodology to detect JavaScript shellcode that fully uses JavaScript code execution environment information with low false negative and false positive. Liu et al. [28] propose a context-aware approach for detection and confinement of malicious JavaScript in PDF by inserting context monitoring code into a document. To analyze JavaScript code with cloaking , Kolbitsch et al. [26] uncover environment-specific malware by exploring multiple execution paths within a single execution. `JSForce` can benefit the dynamic analysis in terms of improved code coverage and tolerance of invalid host environment model.

Researchers also try to combine static and dynamic code features to identify malicious JavaScript programs(Cujo [36]). More precisely, Cujo processes the static program and traces of its execution into q-grams that are classified using machine learning techniques. Symbolic execution [37] is also explored for malicious JavaScript analysis.

Forced Execution: Researchers have proposed to force branch outcomes for different security applications. X-Force [33] can force the binary to execute and explore different execution paths requiring no inputs or proper environment. iRis [19] employs forced execution technique to expose the private API abuses in iOS applications. Forced execution was also proposed to identify kernel-level rootkits] [45], expose hidden behavior in Android apps [24], [44]. To the best of our knowledge, `JSForce` is the first work to enable forced execution on JavaScript for malware detection.

IX. 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 our forced execution engine, the malicious JavaScript detection rate can be greatly improved without any noticeable false positive increase and the runtime overhead was generally neglectable. `JSForce` is made publicly available at [9] as an online service and will release the source code to the security community upon the acceptance for publication.

References

[1] http://www.ecmascript.org/.

[2] http://contagiodump.blogspot.com/2013/03/16800-clean-and-11960-malicious-files.html.

[3] http://malware-traffic-analysis.net/.

[4] http://threatglass.com/.

[5] http://www.alexa.com/topsites.

[6] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[7] https://github.com/Microsoft/ChakraCore.

[8] 2015 symantec internet security threat report. http://www.symantec.com/security_response/publications/threatreport.jsp.

[9] Anonymized for submission.

[10] Sputnik. https://code.google.com/p/sputniktests/.

[11] V8 javascript engine. https://developers.google.com/v8/.

[12] Virus total. https://www.virustotal.com/.

[13] Bandhakavi, S., King, S. T., Madhusudan, P., and Winslett, M. Vex: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium* (2010), vol. 10, pp. 339–354.

[14] Barnard, D. T., and Holt, R. C. Hierarchic syntax error repair for lr grammars. *International Journal of Computer & Information Sciences 11*, 4 (1982), 231–258.

[15] Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., and Yin, H. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. Springer, 2008, pp. 65–88.

[16] Cao, Y., Pan, X., Chen, Y., and Zhuge, J. Jshield: Towards real-time and vulnerability-based detection of polluted drive-by download attacks. In *the Proceedings of Annual Computer Security Applications Conference (ACSAC)* (2014).

[17] Cova, M., Kruegel, C., and 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).

[18] Curtsinger, C., Livshits, B., Zorn, B. G., and Seifert, C. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium* (2011).

[19] Deng, Z., Saltaformaggio, B., Zhang, X., and Xu, D. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 44–56.

[20] FEINSTEIN, B., PECK, D., AND SECUREWORKS, I. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. *Black Hat USA* (2007).

[21] GU, B., ZHANG, W., BAI, X., CHAMPION, A. C., QIN, F., AND XUAN, D. Jsguard: Shellcode detection in javascript. In *Security and Privacy in Communication Networks*. 2013.

[22] HARTSTEIN, B. Jsunpack: An automatic javascript unpacker. In *ShmooCon convention* (2009).

[23] HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming* (1991), Springer.

[24] JOHNSON, R., AND STAVROU, A. Forced-path execution for android applications on x86 platforms. In *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on* (2013), IEEE, pp. 188–197.

[25] KAPRAVELOS, A., SHOSHITAISHVILI, Y., COVA, M., KRUEGEL, C., AND VIGNA, G. Revolver: An automated approach to the detection of evasive web-based malware. In *USENIX Security* (2013), Citeseer, pp. 637–652.

[26] KOLBITSCH, C., LIVSHITS, B., ZORN, B., AND SEIFERT, C. Rozzle: De-cloaking internet malware. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012).

[27] LIKARISH, P., JUNG, E., AND JO, I. Obfuscated malicious javascript detection using classification techniques. In *MALWARE* (2009), Citeseer, pp. 47–54.

[28] LIU, D., WANG, H., AND STAVROU, A. Detecting malicious javascript in pdf through document instrumentation. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on* (2014).

[29] LU, G., AND 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).

[30] MAFFEIS, S., MITCHELL, J. C., AND TALY, A. An operational semantics for javascript. In *Programming languages and systems*. Springer, 2008, pp. 307–325.

[31] MOWERY, K., BOGENREIF, D., YILEK, S., AND SHACHAM, H. Fingerprinting information in javascript implementations. *Proceedings of W2SP 2* (2011).

[32] MULAZZANI, M., RESCHL, P., HUBER, M., LEITHNER, M., SCHRITTWIESER, S., WEIPPL, E., AND WIEN, F. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)* (2013), vol. 5.

[33] PENG, F., DENG, Z., ZHANG, X., XU, D., LIN, Z., AND 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).

[34] POLITZ, J. G., ELIOPOULOS, S., GUHA, A., AND KRISHNAMURTHI, S. Adsafety: Type-based verification of javascript sandboxing. *arXiv preprint arXiv:1506.07813* (2015).

[35] RATANAWORABHAN, P., LIVSHITS, B., AND ZORN, B. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the Usenix Security Symposium* (2009).

[36] RIECK, K., KRUEGER, T., AND DEWALD, A. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference* (2010).

[37] SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010).

[38] SEIFERT, C., WELCH, I., AND KOMISARCZUK, P. Identification of malicious web pages with static heuristics. In *Telecommunication Networks and Applications Conference, 2008. ATNAC 2008. Australasian* (2008), IEEE, pp. 91–96.

[39] TALY, A., ERLINGSSON, Ú., MITCHELL, J. C., MILLER, M. S., AND NAGRA, J. Automated analysis of security-critical javascript apis. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 363–378.

[40] THIEMANN, P. Towards a type system for analyzing javascript programs. In *Programming Languages and Systems*. Springer, 2005, pp. 408–422.

[41] TRINH, M.-T., CHU, D.-H., AND 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* (2014), ACM, pp. 1232–1243.

[42] UPATHILAKE, R., LI, Y., AND MATRAWY, A. A classification of web browser fingerprinting techniques. In *New Technologies, Mobility and Security (NTMS), 2015 7th International Conference on* (2015), IEEE.

[43] WANG, D. Y., SAVAGE, S., AND VOELKER, G. M. Cloak and dagger: dynamics of web search cloaking. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 477–490.

[44] WANG, Z., JOHNSON, R., MURMURIA, R., AND STAVROU, A. Exposing security risks for commercial mobile devices. In *Computer Network Security*. Springer, 2012, pp. 3–21.

[45] WILHELM, J., AND CHIUEH, T.-C. A forced sampled execution approach to kernel rootkit identification. In *Recent Advances in Intrusion Detection* (2007), Springer, pp. 219–235.

[46] XU, W., ZHANG, F., AND ZHU, S. The power of obfuscation techniques in malicious javascript code: A measurement study. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on* (2012), IEEE, pp. 9–16.