J-Force: Forced Execution on JavaScript

Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon,
Yunhui Zheng*, Xiangyu Zhang, Dongyan Xu
Department of Computer Science, Purdue University, USA *IBM T.J. Watson Research Center, USA
{kim1798, kim1634, chungkim, kwon58, xyzhang, dxu}@cs.purdue.edu zhengyu@us.ibm.com

ABSTRACT

Web-based malware equipped with stealthy cloaking and obfuscation techniques is becoming more sophisticated nowadays. In this paper, we propose J-FORCE, a crash-free forced JavaScript execution engine to systematically explore possible execution paths and reveal malicious behaviors in such malware. In particular, J-FORCE records branch outcomes and mutates them for further explorations. J-FORCE inspects function parameter values that may reveal malicious intentions and expose suspicious DOM injections. We addressed a number of technical challenges encountered. For instance, we keep track of missing objects and DOM elements, and create them on demand. To verify the efficacy of our techniques, we apply J-FORCE to detect Exploit Kit (EK) attacks and malicious Chrome extensions. We observe that J-FORCE is more effective compared to the existing tools.

Keywords

JavaScript; Security; Malware; Evasion

1. INTRODUCTION

Web-based applications powered by JavaScript are becoming more widespread, interactive and powerful. In the meanwhile, they are attractive targets of various attacks. Unfortunately, detecting and analyzing malicious web apps against diverse combinations of exploits and evasive techniques is complicated and challenging. Although various detection schemes have been proposed [14, 27, 13], they still suffer from sophisticated attacks such as cloaking attacks [21, 35, 22].

Both static and dynamic approaches have been applied to detect JavaScript malware. Static analysis (e.g., [9, 8]) considers multiple execution paths and usually achieves better code coverage. However, JavaScript is highly dynamic. Static approach may be imprecise and even incapable due to over-approximations and *obfuscations*. This is a critical limitation since obfuscations have been the most common practice to hide the real intentions for protections or malicious reasons. By contrast, dynamic analysis techniques (e.g., [16, 32]) execute the program and thus can reveal concrete behaviors even in an obfuscated program. However, a downside is that they can only cover one concrete execution path in one run and may be unable to hit the spot that conceals malicious behaviors.

To address the limitations, symbolic and concolic execution based techniques [32, 31, 33] have also been proposed to analyze JavaScript programs. While they can generate program inputs and drive the execution along various feasible paths, due to the limitations of the constraint solvers, overcoming *state explosion* and handling *complex JavaScript operations* (e.g., dynamic type conversions, arithmetic/string operations) are still open problems, especially for non-trivial programs built atop various frameworks and other obfuscated programs.

In this paper, we propose J-FORCE, a crash-free[1] JavaScript forced execution engine. J-FORCE *combines* the advantages of static and dynamic approaches: Similar to dynamic analysis, J-FORCE executes the program so that obfuscation is not an obstacle anymore. To increase the coverage, J-FORCE forces the execution to go along different paths. In particular, J-FORCE records the outcomes of branch predicates, mutates them, and explores unvisited paths via multiple executions. This iterative *path exploration* process continues until all possible paths are explored. Hence, J-FORCE can expose not only malicious code that can only be triggered by conditions uneasily met, but also code blocks that are dynamically created and injected. Additionally, J-FORCE further uncovers paths hidden in *event and exception handlers*. J-FORCE can detect evasive attacks triggered by non-deterministic events.

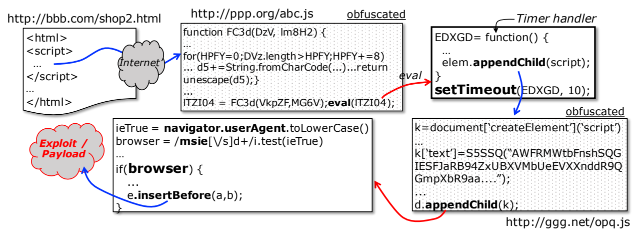
We evaluate J-FORCE on 50 real-world exploits in popular EKs [1, 2] and over 12,000 Chrome extensions. J-FORCE successfully exposed the hidden code of 41 exploits and found that more than 300 Chrome extensions inject advertisements. We also run J-FORCE on 100 JavaScript samples and measure its code coverage capacity. The results show that J-FORCE can cover 95% of the code with 2-8x overhead, which is significantly effective than a popular concolic execution technique (68% coverage, 10-10,000x overhead).

In summary, this paper makes the following contributions.

- We propose J-FORCE, a JavaScript forced execution engine that explores all possible paths to expose hidden malware behaviors. J-FORCE records and switches branch outcomes to explore new paths. J-FORCE unveils function parameter values to detect malicious intentions and DOM injection attacks.
- We address several technical challenges to avoid crashes during the continuous path explorations. For instance, J-FORCE keeps track of missing objects/DOM nodes and creates them on demand. J-FORCE can tolerate critical exceptions and handle infinite loops/recursions.
- We validate the efficacy of J-FORCE through an extensive set of experiments on real-world exploits and web browser ex-

[1]In our paper, *crash-free* is about avoiding or handling JavaScript exceptions.

WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3052674



```
<html>
<script>
 ...
</script>
...
</html>
```

```
function FC3d(DzV, Im8H2) {
...
for(HPFY=0;DVz.length>HPFY;HPFY+=8)
... d5+=String.fromCharCode(...)...return
unescape(d5);}
...
ITZI04 = FC3d(VkpZF,MG6V);eval(ITZI04);
```

```
EDXGD= function() {
  ...
  elem.appendChild(script);
}
setTimeout(EDXGD, 10);
```

```
k=document['createElement']('script')
...
k['text']=S5SSQ("AWFRMWtbFnshSQG
IESFJaRB94ZxUBXVMbUeEVXXnddR9Q
GmpXbR9aa....");
...
d.appendChild(k);
```

```
ieTrue = navigator.userAgent.toLowerCase()
browser = /msie[\/s]d+/i.test(ieTrue)
...
if(browser) {
   ...
   e.insertBefore(a,b);
}
```

Figure 1: Stealthy Exploit Kit Attack.

tensions. J-FORCE successfully disclosed the hidden code in 41 exploits and detected more than 300 ad-injecting extensions. Also, we show that J-FORCE can achieve 95% code coverage and is 2-8x faster than the state-of-the-art on 100 JavaScript samples.

Our work focuses on understanding malicious code that is present on the client, so server-side cloaking or evasion is out-of-scope.

2. MOTIVATION

Recently, Exploit Kits (EKs) have been favored by cybercriminals to perform web-based attacks. In the last year alone, more than 14 attacks were reported to CVE[2]. Since EKs are specially designed to exploit known browser related defects, such attacks are highly effective: once a vulnerable client reaches the actual EK landing page, EK will silently download and install a malware. Therefore, as a defense, it is critical to identify suspicious EK delivery at the first place. Among various delivery vectors, malvertising [10, 37] is one of the most dangerous and successful delivery approaches. In this section, we show a real-world EK delivery equipped with layered obfuscation and cloaking techniques to demonstrate our approach.

Fig. 1 presents a carefully designed multi-layer EK attack chain featured with collaborative cloaking techniques such as *code obfuscation*, *dynamically created scripts* and *evasive paths*: (1) The first obfuscated JS(JavaScript) snippet (`http://ppp.org/abc.js`) is delivered to a legitimate website via malvertising. (2) When it is evaluated during the page loading, it creates a piece of dynamic code from strings using `eval`. (3) The function `EDXGD` in the resulting snippet injects code for the next. Interestingly, `EDXGD` is injected as an event handler and can only be invoked when the timeout event is fired. Once evaluated, the second piece of obfuscated snippet (`http://ggg.net/opq.js`) will be injected into the DOM tree and executed. (4) As a result, another dynamic script is created and injected (`d.appendChild(k)`). (5) The injected code uses a cloaking method to hide the malicious payload: It first checks if the client browser can be the target (`navigator.userAgent` and `msie`). The hidden code is executed only if the check result (`browser`) is true.

Existing Approaches. As two pieces of JavaScript (`abc.js` and `opq.js`) in the chain are obfuscated, static analysis based detection mechanisms [14, 9, 28, 11] may have difficulties in understanding the real semantics and thus are ineffective to handle such cases. Discovering the execution path that can reveal the final exploit payload using dynamic approaches is also difficult. Particularly, it requires invocations of event handlers and proper environment settings (e.g. IE browser), which are conditions not easily met in general. Symbolic and concolic execution techniques [32, 31, 33] can be used to explore multiple feasible paths. However,

[2]CVE-2015-3090, CVE-2015-3105, CVE-2015-5122, CVE-2015-1671, CVE-2015-5119, CVE-2015-5560, CVE-2015-7645, CVE-2015-8651, CVE-2015-8446, CVE-2016-1019, CVE-2016-1001, CVE-2016-0189, CVE-2016-0034, CVE-2016-4117

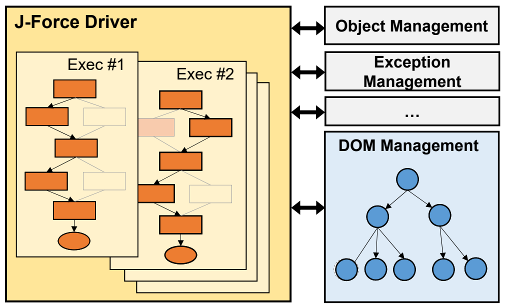


Figure 2: Overview of J-FORCE.

```
<script>
    if (...) {
        btn = document.createElement("button");
        btn.id = "mybutton";
        btn.innerHTML = "Remove";
    } else {
        btn = document.createElement("button");
        btn.id = "mybutton";
        btn.innerHTML = "Skip";
    }
    document.body.appendChild(btn);
</script>
...
<script>
    x = document.getElementById("mybutton");
    if (...) {...}
    if (...) {...}
</script>
```

Figure 3: Example for per-block path exploration.

it is challenging for such techniques to be scalable to complicated and large real-world JavaScript programs due to the limitations imposed by the underlying constraint solvers.

Unfortunately, as shown in Table 1, existing JavaScript malware detection tools are not effective to detect such malware in a scalable way. In particular, while Rozzle [22] performs path explorations on JavaScript programs to reveal evasive malicious behaviors, it cannot disclose code in event handlers as its analysis scope is limited to functions that are explicitly invoked.

J-FORCE Overview. J-FORCE employs a forced execution technique by switching branch outcomes and invoking event handlers. As shown in Fig. 2, J-FORCE explores feasible paths and reveals all the instructions irrespective of branch conditions in multiple concrete executions. Also, event and exception handlers are forcibly invoked without emulating the events. By doing so, J-FORCE is able to reach and expose malicious logic that can only be triggered by a particular combination of events and inputs. Moreover, J-FORCE is dynamic analysis. Hence, it can handle obfuscations and disclose concrete function parameter values, which could further reveal malware behaviors (e.g., identifying `eval` content).

3. DESIGN OF J-FORCE

In this section, we present the details of J-FORCE. We first discuss the J-FORCE execution model. Then we describe how J-FORCE explores multiple execution paths.

3.1 J-Force Execution Model

The execution model of J-FORCE is designed based on the default page rendering model.

3.1.1 Per-block Exploration

The default page rendering order drives the execution of J-FORCE. Once a `<script>` block is evaluated, J-FORCE starts exploring

Table 1: The comparison of the approaches for JavaScript malware detection.

Name	Category	Obfuscation Resilient	Path Exploration Support	State Exploration Free	Events Covered	Exceptions Covered	Target Scope
WebEval [18]	Static & Dynamic Analysis	✓	✗	✓	✗	✗	Chrome Extension
Expector [37]	Dyanamic Analysis	✓	✗	✓	✓	✗	
Hulk [20]	Static & Dynamic Analysis	✓	✗	✓	✓	✗	
Revolver [21]	Static & Dynamic Analysis	✓	✗	✓	✗	✗	Generic
JSAND [13]	Dynamic Analysis	✓	✗	✓	✗	✗	
Nozzle [27]	Dynamic Analysis	✓	✗	✓	✗	✗	
Zozzle [14]	Static Analysis	✗	✗	N/A	✗	✗	
Rozzle [22]	Dynamic (Symbolic Value)	✓	✓	✗	✗	✗	
J-FORCE	**Forced Execution**	✓	✓	✓	✓	✓	**Generic**

all other possible paths within the block. In particular, when J-FORCE reaches the exit of the block, it goes back and explores another unvisited path. Consider the example in Fig. 3. J-FORCE explores the two paths in lines 1-12 before exploring the paths in the next `<script>` block in 14-18.

An alternative is to consider all code blocks as one giant block and explore paths in the "merged" block. However, it can hardly scale because the total number of paths to be explored is the *product* of the path numbers in every individual block, whereas in the per-block strategy it is the *sum* of the number of paths in every block.

Please note that an *external* JS script is essentially a single code block and hence can be explored in a similar way.

3.1.2 Handling Inter-Block Dependencies

One challenge brought by the per-block design is how to consider the dependences across code blocks. For example, in Fig. 3, a same button is set with different texts (`Remove` and `Skip`) along different paths in lines 2-11. Without storing states along different execution paths, our analysis may miss critical states that may lead to malicious behavior. For instance, if we explore the path 7-9 after 2-5. "`Remove`" will be overwritten by "`Skip`" and becomes invisible to blocks afterwards.

While exploring paths globally is the ideal solution, it is unscalable and impractical. Instead, we develop the following technique based on the observation that most inter-block dependences are caused by DOM objects. Since it is valid to have multiple elements with the same name or id on the DOM tree, J-FORCE *allows any DOM injections along any paths*. Also, J-FORCE intercepts relevant DOM APIs (e.g. `getElementById`) and injects choice points, which are conceptually equivalent to `switch-case` statements. So, each execution returns a DOM element (with the same id or name) until all such elements are explored. For example, in Fig. 3, both buttons will be appended to the DOM tree. It further inserts a choice point at line 15. As a result, totally 8 paths are explored in the second block, where 4 are corresponding to the "`Remove`" button and the remaining 4 are for the "`Skip`" button.

In theory, dependencies caused by global variables are handled in the same way. However, it is very expensive to do so for all global variables. Given our focuses are stealthy behaviors that are usually based on string operations, we selectively support global string variables. Furthermore, J-FORCE also overwrites container interfaces (e.g., `hashmap`) to support inserting multiple strings with the same key to a global container. String attributes of DOM objects are handled similarly, where choice points are injected to access the different versions.

3.1.3 Handling Event Handlers

Some event handlers, such as onload, are automatically executed when the corresponding DOM objects are loaded or created. The exploration is driven by the rendering procedure. However, another set of handlers can only be triggered by user and timer events. In our experience, JS malware extensively leverages event handling mechanism to lay out the attack agenda. Fig. 4 shows a simplified step in the malware delivery chain. `__necdel()` is registered as an event handler of `mouseover` event. The script for the next step will not be injected unless the event is triggered. Indeed, we observed many malicious payloads only get triggered by a series carefully organized user or timer events to escape from being detected by honey-client systems or other automatic detection tools. Therefore, exploring event handlers is critical.

```
function __necdel()
{
    var script = document.createElement("script");
    //...
    script.src = "http: //xxx.xxxxxxx.net/";
    var protocol = ("https:" == document.location.protocol: "http://");

    var head = document.getElementsByTagName("head")[0];
    if ((protocol === "http://") && head)
        head.appendChild(script);
}
window.addEventListener("mouseover", __necdel, false);
```

Figure 4: Code injection upon "mouseover" event.

J-FORCE remembers functions registered as event handlers and forces them to be executed. In particular, after the exploration of the current code block, handlers that are registered during exploration are executed, without requiring the triggering events. The individual handlers are considered as code blocks that are explored separately. To the best of our knowledge, most existing honey-client systems and JS symbolic execution engines (e.g, [31]) do not emulate events. Hence, they cannot reveal sophisticated handler-related behaviors.

3.1.4 Handling Asynchronous Execution

Currently, J-FORCE does not focus on exposing race conditions caused by asynchronizations [29, 38]. In fact, most JS races are transient [24]. In our experience, we have not observed any real-world malicious attacks leveraging race conditions due to its non-deterministic and unreliable nature.

J-FORCE respects browser's decision on which block runs first. Note that JavaScript execution is single threaded and the execution of a code block cannot be interrupted. J-FORCE only steps in when a block is being evaluated for the purpose of per-block code exploration.

3.1.5 Handling Dynamic Code Evaluation

JavaScript is highly dynamic. Malicious JS snippets can be dynamically created from strings. For example, a common practice is to create a `<script>` element, specify its source and attach it to the DOM tree. `eval()` is another way to run dynamic code.

J-FORCE admits all code injections found along different paths during the path exploration. Consequently, they will be explored like other code on the DOM tree. Some code snippets may be added to DOM elements that have already been rendered and explored by J-FORCE. For such cases, J-FORCE restarts the rendering procedure but only explores the uncovered injected snippets.

For code dynamically evaluated by functions like `eval`, J-FORCE explores the code snippet concealed in the function parameter, as a part of the parent code block exploration. Note that J-FORCE provides versioning support for strings so that different but concrete parameter values produced by previous logic will be explored.

3.2 Path Exploration

J-FORCE explores different paths in multiple runs. In each run, it looks for opportunities where mutating a predicate leads to unexplored instructions. Once found, it *forces* the execution to cover them in future iterations. It repeats this procedure until all instructions are covered. We designed two exploration strategies depending on the needs.

- *L-path* executes each instruction at least once with linear time complexity. Exploring all distinct paths is not its priority. For JS malware analysis, this strategy is sufficient in most cases as malicious behaviors are usually hidden in blocks.
- *E-path* aims at exploring all possible execution paths with exponential time complexity. We observed that only a few advanced malware examples requires the *E-path* strategy.

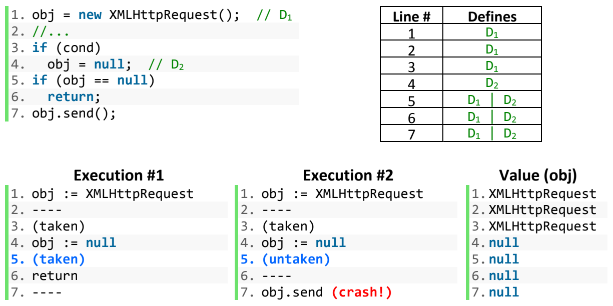
Algorithm 1 Path Exploration.

```
Input: $\mathcal{I}$: JavaScript instructions in a program
     // $\sigma$ is a list of forced predicates. A predicate $p$ is represented as a tuple
     // $(p_{src}, p_{dst})$ that specifies the source $src$ and forced target $dst$
 1: function ForcedExec($\sigma$)
 2:     $\sigma_e \leftarrow [\,]$     // $\sigma_e$ is a list of executed predicates
 3:     $p \leftarrow$ Pop_Front( $\sigma$ )
 4:     for each $i$ in $\mathcal{I}$ do
 5:         if $i$ is a condition branch instruction then
 6:             if $i_{src} \equiv p_{src}$ then     // $i_{src}$: source address of $i$
 7:                 $i_{dst} \leftarrow p_{dst}$     // specify the instruction to be executed
 8:                 $p \leftarrow$ Pop_Front( $\sigma$ )
 9:             else
10:                 $E \leftarrow E \cup \{i_{dst}\}$
11:             $\sigma_e \leftarrow \sigma_e \cdot (i_{src}, i_{dst})$
12:         Execute the instruction $i$
13:     return $\sigma_e$

14: function PathExploration( )
15:     $E \leftarrow \{\}$ // explored instructions
16:     $W \leftarrow \{$ForcedExec$(nil)\}$ // initial execution. $W$: worklist
17:     while $W \neq \varnothing$ do
18:         $\sigma' \leftarrow$ Pop$(W)$
19:         $\sigma_t \leftarrow nil$
20:         for each $p$ in $\sigma'$ do
21:             if HasAnyUnexploredTarget$(E, p)$ then
22:                 $\sigma'_t \leftarrow \sigma_t \cdot$ SwitchingTarget$(p)$
23:                 $W \leftarrow W \cup \{$ForcedExec$(\sigma'_t)\}$
24:             else
25:                 $\sigma_t \leftarrow \sigma_t \cdot p$
```

Algorithm 1 shows the details of the path exploration approach. Function FORCEDEXEC explains how to drive the execution to a desired branch. In particular, it takes a forced execution schema σ as the input. σ is a list of tuple (p_{src}, p_{dst}), where p_{src} is the address of a predicate p and p_{dst} is the forced target. Intuitively, it specifies the next step (p_{dst}) when J-FORCE sees p. The logic of forced execution is specified in the loop starting at line 4 interpreted by JS engine. If a rerouting schema is provided for the current branching instruction i (line 6), J-FORCE forces the execution to take the branch specified in the scheme at line 7. Otherwise, the instruction will be executed normally.

Function PATHEXPLORATION is the top-level driver. It maintains a worklist *W*, which is a set of forced execution schemes. E is a set of covered instructions. J-FORCE uses it to discover unexplored instructions. At line 16, J-FORCE starts the execution with no forced execution scheme and just runs the whole program normally. The purpose of this step is to obtain a list of predicates on one path. Then, J-FORCE can develop a new scheme by mutating a predicate at line 22 to execute uncovered instructions (line 21). The driver repeats this until the worklist is empty, meaning that no further opportunities can be discovered. Although the exploration algorithm stems from *L-path* strategy, *E-path* takes the same phase except at line 21. Particularly, at the given branch, instead of checking if its feasible targets are disclosed, *E-path* makes sure the branch is followed along with two different targets.



```
1. obj = new XMLHttpRequest();  // D₁
2. //...
3. if (cond)
4.   obj = null;  // D₂
5. if (obj == null)
6.   return;
7. obj.send();
```

Line #	Defines
1	D_1
2	D_1
3	D_1
4	D_2
5	D_1 | D_2
6	D_1 | D_2
7	D_1 | D_2

```
Execution #1                Execution #2                Value (obj)
1. obj := XMLHttpRequest    1. obj := XMLHttpRequest    1.XMLHttpRequest
2. ----                     2. ----                     2.XMLHttpRequest
3. (taken)                  3. (taken)                  3.XMLHttpRequest
4. obj := null              4. obj := null              4.null
5. (taken)                  5. (untaken)                5.null
6. return                   6. ----                     6.null
7. ----                     7. obj.send (crash!)        7.null
```

Figure 5: Handling crashes caused by missing objects.

4. CRASH-FREE FORCED EXECUTION

As J-FORCE ignores path conditions, a program may execute along an infeasible path and crash. In this section, we describe the challenges and our solutions to avoid crashing.

4.1 Missing Object

Fig. 5 shows a typical example of the crashes caused by missing objects. At line 1, variable `obj` is initialized to an Ajax object. Suppose the `true` branches of the two predicates (line 3 and 5) are taken in the first run. Since line 7 is not explored, in the second run, the predicate at line 5 is mutated. However, as `obj` has been set to `null` at line 4, the program will crash at line 7.

To handle this, when resolving an object accessed, J-FORCE first identifies a set of candidates, which can be collected using an existing data flow analysis. In addition, candidates without correct properties and types are filtered out. As shown in the defines table in Fig. 5, at line 7, $\mathtt{D_1}$ and $\mathtt{D_2}$ are possible objects to be accessed. However, only $\mathtt{D_1}$ has the correct field `send`. Therefore, J-FORCE selects $\mathtt{D_1}$ and continues the forced execution.

4.2 Handling Missing DOM Elements

Another common kind of crashes in forced execution is caused by missing DOM elements. Our strategy is to create and insert the missing ones to the DOM tree *on demand*. Note that simply creating a new DOM element on each access without appending it to the right place will not work in practice. If multiple accesses to a same element yield different newly created objects, the program semantics will be violated. However, as DOM elements can be selected in various ways (e.g., by id, XPath, etc.), the challenge lies in how to put the new elements in the right place.

If the selection is by element id, name, tag and class, the solution is straightforward. Particularly, as shown in Algorithm 2, if the element returned by the original selector is invalid (line 4), J-FORCE creates a new one and inserts it to the children list of the current element (line 8-9).

Handling XPath selectors is more challenging. An XPath may be fully specified (e.g., "`/A/B/C`" means `C` is an immediate child of `B` and `B` is a child of `A`) or partially specified (e.g., "`/A//C`" means

```
1. if (window.attachEvent) {
2.   window.attachEvent("onload", window["load" + initialize]);  // ...
3. } else {
4.   window.addEventListener("load", initialize, false);  // ...
5. }
```

Figure 6: Browser-compatibility exception in forced execution.

all `C` objects with an ancestor `A`). An XPath may also contain wildcards to select all elements satisfying the filtering conditions (e.g., "`/A[@exchange]`" selects `A` with attribute `exchange`). In a forced run, an XPath selector may be partially broken due to missing elements. Consider selector "$p \cdot s$". The prefix p correctly locate a DOM element. However, the suffix s fails because there is no such elements. To handle this issue, J-FORCE identifies the longest p that can locate a valid element o, creates element(s) corresponding to s and make them a subtree of o.

Function `PathRecognizer()` in Algorithm 2 describes the procedure. Particularly, at line 13, an XPath p is split by delimiters (i.e., '`/`' and '`//`'). Each delimited segment τ contains three parts: (1) the delimiter τ_p ("", "`/`" or "`//`"); (2) the id τ_e (e.g., `A`), and (3) the filter τ_a (e.g., `[@exchange]`).

If τ_p is "`//`", `GetOffSpring` is invoked to identify the offsprings of the current object θ that matches τ_e and τ_a (line 22). Otherwise, `GetChildren` is called to get the direct children of the current object that matches τ_e and τ_a (line 16). If no element is found (line 19), a new element corresponding to τ_e and τ_a is created as a child of θ (line 20). The above procedure continues until the original selector becomes valid.

An important design choice made is that the elements created during one (forced) run are retained for later executions. This avoids creating duplicated elements in multiple executions and the DOM tree grows monotonically. In practice, we found the size of a DOM tree usually increases slowly and gradually becomes stable.

Algorithm 2 Handle missing DOM elements.

```
Input: $\sigma \in \{id, name, name_{tag}, name_{class}, \text{XPath}\}$
 1: function CHECKANDINSERTION($\sigma$)
 2:   $E \leftarrow$ GETELEMENTS($\sigma$)
 3:   $\tau \leftarrow$ GETCURRNETOBJECT()
 4:   if $\neg$ ISVALID($E$) then
 5:     if $\sigma \in$ XPath then
 6:       return PATHRECOGNIZER($\sigma$)
 7:     else
 8:       $\tau$.INSERT(CREATEELEMENT($\sigma$))
 9:       $E \leftarrow$ GETELEMENTS($\sigma$)
10:   return $E$
11: function PATHRECOGNIZER($p$)
12:   $\theta \leftarrow$ the current node
13:   $p' \leftarrow$ PARTITIONBYDELIMITER($p$)
14:   for each segment $(\tau_p, \tau_e, \tau_a)$ in $p'$ do //$\tau_p$:delimiter, $\tau_e$:identifier, $\tau_a$:filter
15:     if $\tau_p \equiv$ "//" then
16:       $E \leftarrow \theta$.GETOFFSPRINGS($\tau_e, \tau_a$)
17:     else   /*$\tau_p \equiv$ '/' $\vee$ $\tau_p$ is empty*/
18:       $E \leftarrow \theta$.GETCHILDREN($\tau_e, \tau_a$)
19:     if $\neg$ ISVALID($E$) then
20:       $\theta$.INSERT(CREATEELEMENT($\tau_e, \tau_a$))
21:       $E \leftarrow \theta$.GETCHILDREN($\tau_e, \tau_a$)
22:     $\theta \leftarrow E$
23:   return $E$
```

4.3 Handling Exception

Being able to recover from crashes caused by exceptions is one of the most important features of J-FORCE for robustness. As the program may be forced to run on an infeasible path, various exceptions may occur. For example, Fig. 6 shows a common practice to make the program compatible with different browsers. J-FORCE will execute line 2 without considering its predicate and thus triggers an exception. Since the corresponding handler is absent, the forced execution will be interrupted and terminated.

```
1. if (...) {
2.   var script = document.createElement("script");
3.   script.src = "http://.../a.js";
4.   document.body.appendChild(script);
5. } else {
6.   window.location = "http://.../b.html";  /* page redirection */
7. }
```

Figure 7: An example of page redirections.

To avoid terminations due to such exceptions, J-FORCE captures all unhandled exceptions using a top-level exception handler in the global scope and resumes the interrupted execution from the nearest legacy function by unwinding the stack. In addition, to preserve the semantics of the exception triggering statement, J-FORCE includes a set of selective legacy APIs, which will be invoked based on the context. For instance, in Fig. 6, the `attachEvent` is redirected to the `addEventListener` so that the original program semantics are preserved. Algorithm 3 explains the details:

(a) *Exceptions that can be handled by the original program*: J-FORCE remembers the triggering location (line 3) and then explores the corresponding `catch` block. The code after the triggering point will be covered in a later iteration.

(b) *Uncaught exceptions due to missing handlers*: They will be taken care of by the top-level handler inserted by J-FORCE (lines 6,7-12).

(c) *Exception handlers present but no exception was triggered in one run.* In our experience, a `catch` block is a high-value target for exploration, as malware authors often place their malicious code here for cloaking [22, 21]. These handlers hence should be explored regardless the exception occurences: J-FORCE employs the same strategy for (a). J-FORCE remembers the block entry point and explores it later.

Algorithm 3 Exception Handling.

```
 1: function EXCEPTIONOCCURENCE($\sigma$)
 2:   if ISCOUGHT($\sigma$) then
 3:     SAVEEXCEPTIONLOC($\sigma$)
 4:     return // Allow to run catch block
 5:   else
 6:     return TOPLEVELHANDLER($\sigma$)
 7: function TOPLEVELHANDLER($\sigma$)
 8:   $t \leftarrow$ FINDLEGACYFUNC($\sigma$)
 9:   if ISVALID($t$) then
10:     return CALL($t$)
11:   else
12:     return and allow to run the following.
```

4.4 Page Redirection

Page redirections are commonly used to send visitors to a new destination by setting the `location` attribute of the `window` object in JavaScript. A page redirection cancels the current page rendering procedure (including the JavaScript execution and resource downloading) and hence interrupts J-FORCE's code exploration strategy (J-FORCE explores paths in multiple runs).

Fig. 7 shows an example. The `true` branch of the `if` statement injects a new `<script>` element while the `else` branch redirects visitors to `b.html`. Consider the following forced execution. In the 1^{st} run, the `true` branch is covered and a new piece of JavaScript in `a.js` will be downloaded and executed (lines 2-4). (`a.js`). As explained in the forced execution model, J-FORCE explores the current code block before processing the next block. Hence, in the next iteration, it explores the `else` branch before executing `a.js`. However, since the page redirection happens at line 6, the forced execution will be interrupted so that `a.js` will not be explored. In fact, if there are other uncovered paths/blocks in the same page, they will not be explored due to the page redirection.

Our solution is to load the target page in a separate frame so that J-FORCE can continue exploring the current page. Since frames are isolated from each other, the effect of loading the destination page in a frame is functionally equivalent to a page redirection. In this particular example, J-FORCE loads `b.html` in an `iframe` and thus is able to explore the behaviors in `a.js`.

4.5 Infinite Loop and Recursion

J-FORCE may suffer from infinite loops or endless recursions because it ignores the loop and recursion conditions. To handle this issue, we set an upper bound on the number of times a loop or a recursive function can be invoked. For loops, J-FORCE monitors the loop executions and makes sure that they do not go beyond the threshold. Otherwise, J-FORCE forces the execution to skip the loop. Similarly, for recursions, we use a threshold to limit recursion depth. We make sure that whenever new stack frame is created, the stack depth is smaller than the threshold.

5. EVALUATION

J-FORCE is implemented atop WebKit-r171233 with GTK+ port. Our evaluation consists of two experiments. The first one is a systematic study on 50 EK samples and 12, 132 Chrome extensions to see if J-FORCE is able to detect (malicious) behaviors covered by sophisticated cloaking and obfuscation techniques. Also, since being able to explore more code is important, in the second experiment, we further quantify J-FORCE's performance by measuring the coverage and the overhead on 100 real-world JavaScript programs. All experiments are performed on a machine with an Intel Core i7 3.40 GHz CPU and 12 GB RAM running Ubuntu 14.04 LTS.

5.1 Detecting Suspicious Hidden Behaviors

5.1.1 Detecting Obfuscations and Evasions in EKs

We have collected 50 EK samples from various sources [1, 2], and classified them based on the underlying EKs, namely Angler, RIG, Nuclear, Magnitude, SweetOrange. Although different, we observed they all share similar mechanisms listed as follows:

- *Obfuscation.* Obfuscation conceals program functionalities using string operations to make detecting malware challenging. In EK, obfuscation technique is used more than once throughout multiple layers of code injection.
- *Evasion.* To minimize the possibility of being caught (e.g., by honey-pot based approaches), EK only invokes the malicious logic when it satisfies certain conditions. Specifically, EK usually scans visitors' system (e.g. the signatures of browsers, extensions, etc.) before moving on to the next stage. An example is shown in Fig. 1 in Sec. 2.
- *Exploiting Vulnerabilities.* EK is designed to exploit particular vulnerabilities in browsers or add-ons by hijacking the control flow and elevating permissions. The typical targets of such exploitation are Adobe Flash, MS Silverlight and Java runtime as well as browsers themselves.
- *Payload Delivery.* As the last step, a malicious binary is downloaded and executed without user's consent. Ransomware [7] and click fraud [6] are two common examples.

As J-FORCE focuses on detecting malicious JavaScript behaviors, only the JavaScript parts (`obfuscation` and `evasion`) are included for evaluation. Analyzing non-JavaScript code, such as exploiting vulnerabilities in the web browser or plug-ins, is beyond the scope of this paper. The results of experiments on 50 EK samples (10 for each EK type) are presented in Table 2. It shows the number of the samples can be handled by each tool, in terms of *obfuscation handled* and *evasion passed*. Since we know the ground truth about *deobfuscation*, counting successful de-obfuscations is straightforward. For *evasions*, if the exploitation entry point (e.g. `<object>`) is reached, we say the evasion is detected.

Exploit Kits	# of samples	# of samples whose *obfuscations* / *evasions* can be handled			
		Native run	Rozzle [22]	WebEval [18]	J-FORCE
Angler	10	2 / 1	7 / 6	3 / 3	10 / 10
RIG	10	5 / 0	7 / 2	5 / 0	10 / 10
Nuclear	10	3 / 0	6 / 2	3 / 1	10 / 7
Magnitude	10	6 / 2	10 / 6	6 / 4	10 / 10
SweetOrange	10	2 / 0	8 / 4	4 / 4	10 / 6

Table 2: Comparing detection techniques on EKs.

	# of Ad-injecting			# of Info. leakage		
	Total	Ajax	Script Injection	Total	Ajax	Script Injection
Hulk [20]	195	29	166	14	9	5
Expector [37]	187	28	159	9	6	3
WebEval [18]	158	15	143	8	5	3
J-FORCE	322	45	277	30	21	9

Table 3: The analysis result of 12,132 Chrome extensions.

The results show that J-FORCE is able to handle more obfuscations and evasions than others, hence can expose more hidden malicious behaviors in EK attacks. In particular, J-FORCE is significantly effective in detecting evasions. While J-FORCE outperforms other techniques, it misses a few evasions in *Nuclear* and *SweetOrange*. We manually inspected these cases and found that they use Visual Basic (VB) scripts which are not currently supported by J-FORCE. However, our design is general and can be implemented on VB scripts too.

5.1.2 Detecting Ads Injections in Chrome Extensions

Browser extensions are commonly used nowadays to enhance user experience and thus becoming a target of adversaries. Several recent work [20, 18, 37] have been proposed to analyze extensions. In this section, we show how J-FORCE can effectively disclose suspicious behaviors in Chrome extensions.

We crawled and obtained 12,132 extensions from Chrome Web Store [5] in July 2016. The analysis is done offline. As the JavaScript APIs used in extensions are slightly different from those in web applications, we enhance J-FORCE to support such Chrome APIs (e.g., `chrome.browserAction.onClicked`). In this experiment, we are particularly interested in detecting ad-injections and information leaks. We also compare with recent work on Chrome extension analysis [20, 18, 37].

Table 3 summarizes the experiment results. J-FORCE detected 322 extensions that inject advertisement, where 277 deliver ad contents using script injections and the remaining ones bring in ads via Ajax. Comparing to other techniques, J-FORCE is able to find 195 more ad-injecting extensions, which confirms its effectiveness of handling cloaking and fingerprinting techniques. In addition, J-FORCE detected 30 extensions that send out sensitive information such as passwords and cookies via Ajax, while other techniques can detect at most 14 of them.

Table 4 presents the statistics of the Chrome extension execution analysis. We report the minimum, average and maximum number of JavaScript IR instructions, script injections, Ajax requests, `eval` function invocations, event handlers and page redirections observed in exploring one extension. The results show that J-FORCE can exercise more instructions and discover more behaviors than the native run. We also report the number of runs required by J-FORCE to cover all instructions (using the *L-path* search strategy explained

	JavaScript IR			Script Injections			Ajax			Eval			Event Handlers			Redirections			Handled Crashes			# of Runs		
	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max
J-FORCE	1,478	10	31,248	0.71	0	28	0.21	0	5	0.27	0	10	1.57	0	19	0.15	0	5	2.74	0	117	11.32	1	609
Native run	406	10	14,151	0.46	0	13	0.03	0	2	0.15	0	8	0.85	0	12	0.02	0	2	N/A			N/A		

Table 4: The statistics of Chrome extensions analysis

in Sec. 3.2). We show the number of potential crashes caused by the forced execution. We observed 2.74 crashes per extension on average and they are mostly caused by missing objects and DOM elements. All of them are handled correctly using the approach discussed in Sec. 4.

5.1.3 Case Study - Anti-adblocker

Unlike traditional programs, web applications have various external dependences. For example, they can navigate the execution depending on browsers environment settings. They can download and load different external JavaScript on the fly from third parties during executions. Therefore, although it is possible mutating input values may change the execution paths, in general, it is highly nontrivial or even infeasible for an automatic exploration tool to satisfy the triggering conditions of the execution environment and third party scripts. In this case study, we showcase a real-world anti-adblocker [4] to demonstrate how J-FORCE bypasses sophisticated predicates and thus can be helpful for understanding stealthy program behaviors.

Ad-blocker (e.g., [3]) is a piece of software that allows clients to roam the web without encountering any Ads. In particular, it utilizes network control and in-page manipulation to help users block advertisements loaded from ad-network. As many content publishers make their primary legitimate income from Ads, there are growing demands for delivering ads even the ad-blockers are running in client browsers. As a result, anti-adblockers have been developed and deployed by publishers on their websites. Anti-adblockers are usually scripts delivered by publishers to detect if adblockers are enabled in the client browsers. Once found, it either hides the content or delivers the ads by circumventing the ads filters.

Fig. 8 presents a simplified version of a popular anti-adblocker *BlockAdblock* [4], where the arrows denote important call edges. It first detects if an adblocker is enabled on the client-side and loads the real ads contents that are delivered as an image. In particular, line 1 includes an external script ("*advertising.js*"). If it can be successfully loaded, variable `__haz` will be set to `false`. If an adblocker presents, the script will not be blocked and the value of `__haz` remains `undefined`. Therefore, *BlockAdblock* can tell if an adblocker is running by checking the value of `__haz`. At line 4, it invokes function `__ac()` and defines the function to be invoked for the next step. Depending on the presence of an adblocker, it will invoke a function (defined in lines 13-23) or do nothing. In function `__dec`, it loads an image, where its URL is specified at line 3 and further transformed at line 4. Interestingly, instead of displaying the image, it uses this image as a circumvention of ad-blocking rules and loads the raw data of the images. At line 21, function `__cb` is invoked, which creates a `div` element and displays the HTML hidden in the image at line 27.

It is highly nontrivial for static analysis based approaches to precisely analyze such complicated call relations, as it requires advanced alias and string analysis (e.g., the operations in line 4 and 20). More importantly, as the ads contents are actually hidden in an image, they may not even be in the analysis scope. As a result, it is very unlikely that the static analysis can handle such cases. Another option is to actually run the program. However, one important triggering condition of the secret loading procedure is that the external script included at line 1 must be blocked by an adblocker, which is highly dependent on the execution environment. If the adblocker has not been configured correctly or the URL of the external resource is not on the blacklist anymore, dynamic analysis cannot unveil the stealthy operations either.

By contrast, J-FORCE decouples the dependencies on the environment and hence allows us to effectively and deterministically observe unusual behaviors. On the left hand side of Fig. 8, we compare the control flow graphs that highlight the differences between J-FORCE and dynamic analysis based approaches. J-FORCE is able to explore both paths while the dynamic analysis only covers one path. As such, J-FORCE is able to discover the real ads contents by forced execution without requiring complicated system settings to actually trigger the logic in traditional dynamic approaches.

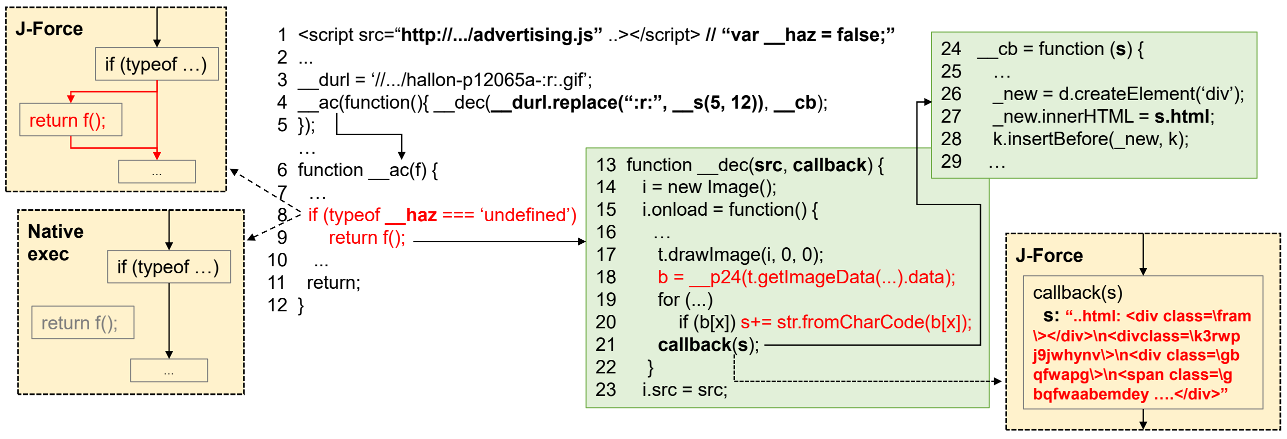
More importantly, through J-FORCE, we can uncover the actual values of function parameters (the right side of Fig. 8) and track the origin of suspicious values. With such capabilities (especially the hidden contents that can only be obtained dynamically), it is straightforward to conclude the ads are included in the image file.

5.2 Efficiency

As described in Sec. 3.2, J-FORCE can be configured to improve coverage on instructions (the *L-path* strategy) or paths (the *E-path* strategy). To measure its efficiency, we extracted 100 examples (from Alexa.com) and evaluate J-FORCE on these real-world JavaScript programs. We compare J-FORCE with Jalangi, a *concolic* JavaScript execution engine [32], which is one of the closest alternate approaches available at present.

Fig. 9 presents the code coverage comparison results. The number of branches of the benchmarks varies from 109 to 1,200. In Fig.9, the JavaScript benchmarks on the X-axis are sorted by the branch count in ascending order. The result shows that, on average, J-FORCE is able to cover 95% of the code (the same result for both exploration strategies), which is significantly more than Jalangi (less than 68%). We found that the main reason for the improvement is that the concolic execution based approach does not explore the code in event and timer handlers. In addition, Jalangi often fails to handle complex arithmetic operations such as division and modulo. By contrast, J-FORCE does not suffer from such limitation and is able to expand its analysis scope to event and exception handlers. Besides, J-FORCE does not miss conditional blocks as our exploration technique is designed to cover both branches by switching branch outcomes. We also manually inspect the scenarios where J-FORCE fails to cover all instructions. We found that this is mainly due to coding errors in the sample JavaScript programs.

Beside the coverage, we also measure the runtime performance of J-FORCE. Fig. 10 summarizes the comparison result of the overheads collected during the coverage test. For each approach, the overhead is normalized to the native run. The result shows that the overhead of J-FORCE is 2-8x (2-300x for *E-path*) whereas Jalangi has much higher overhead 10-10,000x. Observe that such a difference is caused by the fact that concolic execution based approaches may not scale well with the number of branches, showing exponentially increasing overhead. Particularly, generating and solving path constraints is more expensive than mutating branch outcomes.



```
<script src="http://.../advertising.js" ..></script> // "var __haz = false;"
...
__durl = '//.../hallon-p12065a-:r:.gif';
__ac(function(){ __dec(__durl.replace(":r:", __s(5, 12)), __cb);
});
…
function __ac(f) {
  …
  if (typeof __haz === 'undefined')
    return f();
  …
  return;
}
```

```
function __dec(src, callback) {
  i = new Image();
  i.onload = function() {
    …
    t.drawImage(i, 0, 0);
    b = __p24(t.getImageData(...).data);
    for (...)
      if (b[x]) s+= str.fromCharCode(b[x]);
    callback(s);
  }
  i.src = src;
```

```
__cb = function (s) {
  …
  _new = d.createElement('div');
  _new.innerHTML = s.html;
  k.insertBefore(_new, k);
…
```

```
s: "..html: <div class=\fram
\></div>\n<divclass=\k3rwp
j9jwhynv\>\n<div class=\gb
qfwapg\>\n<span class=\g
bqfwaabemdey ….</div>"
```

Figure 8: Analyzing Anti-Adblocker using J-Force.

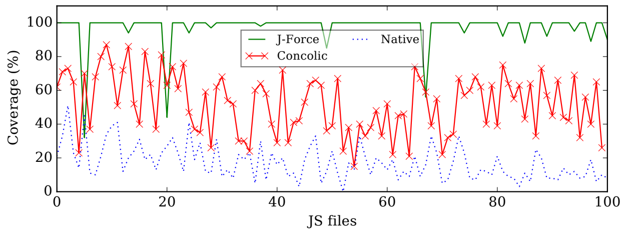


Figure 9: Coverage of J-Force in comparison with native run and concolic execution.

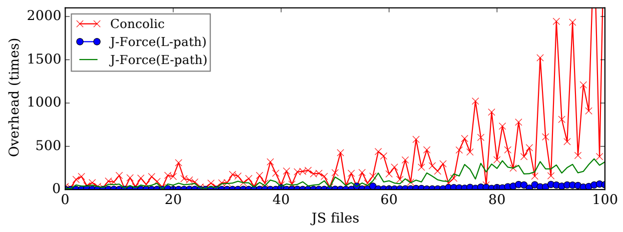


Figure 10: Performance overhead of J-Force in comparison with concolic execution.

6. RELATED WORK

Multiple Path Execution. The concept of forced execution was employed in previous researches [26, 15, 36, 19]. Although the concept has been applied in various domains, such as native binary programs [26], mobile apps [15, 19], and identifying kernel rootkits [36], our work is the first to propose the forced execution engine for JavaScript to the best of our knowledge. Furthermore, the challenges that J-Force solves, such as handling missing objects/DOM, handling event/exception handlers and more (Sec. 4) are unique to JavaScript and are not proposed (or solved) by previous work. Rozzle [22] also places emphasis on analyzing self-revealing program behaviors. It explores multiple execution paths with single execution. However, it is done via a different approach which is based on symbolic values. More importantly, they have limited support for program faults and exceptions handling. By contrast, our tool can explore all feasible paths without being interrupted by exceptions. Symbolic (or concolic) execution has been applied to analyze JavaScript based Web applications [32, 31, 33]. Due to the limitations in underlying constraint solvers, it is challenging to support dynamic nature and scale to real-world applications built atop various JavaScript frameworks.

JavaScript Malware. EVILSEED [17] leverages characteristics of known malicious web pages to discover other likely malicious web pages including JavaScript. Revolver [21] aims to find JavaScript malware based on code similarity. In particular, it tries to classify evasive malware by comparing with a large amount of JavaScript collected in advance. It heavily resorts to the result of pre-classification by oracle, and may not be robust against newly crafted malware (e.g., zero-day exploit). MineSpider [34] extracts URLs from JS snippets equipped with evasion techniques that performs drive-by download attacks. It collects execution paths relevant to redirections using program slicing methods. While it is useful to track page redirections, it is not able to handle the dynamic remote code injection using iframe or simple `<script>` tag. Lekies et al. [23] show attack methods enabled by the object scoping and dynamic nature of JavaScript. They investigate a set of high-ranked domains and verify that those are vulnerable to Cross-Site Script Inclusion(XSSI) attacks. ScriptInspector [39] examines third-party script injection to restrict accesses to critical resources. This is achieved by allowing site administrators to establish their own security policies. WebCapsule [25] records and replays web contents executions for forensic analysis. It records and all non-deterministic inputs to the core web rendering engine including user interactions. RAIL [12] can verify security patches of web applications by rerunning patched web applications with previous buggy inducing inputs such as exploits. The system can tolerate state divergences caused by the patches. Unlike the record and replay approaches, J-Force explores all possible paths to reveal evasive malicious logics which are difficult to expose.

Browser Extensions. Hulk [20] analyzes Chrome browser extensions and detects malicious (or suspicious) behaviors, such as ad-injecting and information leak. Expector [37] tries to figure out the correlation between malvertising and plug-ins. It shows that, in a condition where a specific extension is working, malvertising is more likely to appear. WebEval [18] inspects Chrome extensions upon the combination of static and dynamic analysis. In order to trigger malicious activities, it sets up simulations by recording complex interactions between web pages and network events. Observe that though such techniques have their own way to increase coverage and unveil hidden malicious actions, it would not be sufficient to induce all possible behaviors.

7. DISCUSSION

As our solution aims to expose malware hidden under a certain program path, detecting data driven attacks is still challenging. Although diverting control flow by the forced execution occasionally breaks the program semantics, due to the stealthy pattern and conditional nature of the hidden code, we are confident that J-FORCE is able to disclose most of evasive malware in the wild. Since J-FORCE is currently designed to detect client-side JavaScript malware, handling cloaking schemes in the server-side scripts (e.g. SQL, PHP, etc. [30]) is beyond the scope of this paper.

8. CONCLUSION

In this paper, we proposed J-FORCE, a forced execution engine for JavaScript to expose hidden and even malicious program behaviors. J-FORCE explores all possible execution paths by mutating the outcomes of branch predicates. We solved multiple technical challenges and make J-FORCE a practical, robust and crash-free tool. We validate the efficacy of J-FORCE through an extensive set of experiments. J-FORCE has been evaluated on 50 exploits of popular exploit kits and more than 12,000 Chrome extensions. It successfully unveiled the hidden code in 41 exploits and detected more than 300 Chrome extensions injecting advertisements. The experiments on 100 real-world JavaScript samples show that J-FORCE is able to achieve 95% code coverage and perform 2-8x better than existing approaches.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1409668, 1320444, and 1320306, ONR under contract N000141410468, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

10. REFERENCES

[1] http://malware.dontneedcoffee.com.

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

[3] Adblock plus. https://adblockplus.org.

[4] Blockadblock. http://blockadblock.com.

[5] Chrome Web Store. https://chrome.google.com/webstore.

[6] Clickfraud. http://digitalmarketingmagazine.co.uk/digital-marketing-advertising/the-crooks-willing-to-put-you-out-of-business-for-5/1740.

[7] Cryptolocker: What is and how to avoid it. http://www.pandasecurity.com/mediacenter/malware/cryptolocker/.

[8] JSHint. http://jshint.com.

[9] JSLint. http://www.jslint.com.

[10] Malvertising, Exploit Kits, ClickFraud & Ransomware: A Thriving Underground Economy. https://www.zscaler.com/blogs/research/malvertising-exploit-kits-clickfraud-ransomware-thriving-underground-economy.

[11] Y. Cao, X. Pan, Y. Chen, and J. Zhuge. Jshield: towards real-time and vulnerability-based detection of polluted drive-by download attacks. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 466–475. ACM, 2014.

[12] H. Chen, T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Identifying information disclosure in web applications with retroactive auditing. In *OSDI*, pages 555–569, 2014.

[13] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, pages 281–290. ACM, 2010.

[14] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*, pages 33–48, 2011.

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

[16] L. Gong, M. Pradel, M. Sridharan, and K. Sen. Dlint: Dynamically checking bad coding practices in javascript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 94–105. ACM, 2015.

[17] L. Invernizzi and P. M. Comparetti. Evilseed: A guided approach to finding malicious web pages. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 428–442. IEEE, 2012.

[18] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and lessons from three years fighting malicious extensions. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 579–593, 2015.

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

[20] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *Proceedings of the 23rd Usenix Security Symposium*, 2014.

[21] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *USENIX Security*, pages 637–652. Citeseer, 2013.

[22] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 443–457. IEEE, 2012.

[23] S. Lekies, B. Stock, M. Wentzel, and M. Johns. The unexpected dangers of dynamic javascript. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 723–735, Washington, D.C., Aug. 2015. USENIX Association.

[24] E. Mutlu, S. Tasiran, and B. Livshits. Detecting javascript races that matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 381–392, New York, NY, USA, 2015. ACM.

[25] C. Neasbitt, B. Li, R. Perdisci, L. Lu, K. Singh, and K. Li. Webcapsule: Towards a lightweight forensic engine for web browsers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 133–145. ACM, 2015.

[26] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-force: Force-executing binary programs for security applications. In *Proceedings of the 2014 USENIX Security Symposium, San Diego, CA (August 2014)*, 2014.

[27] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, pages 169–186, 2009.

[28] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM, 2015.

[29] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *ACM SIGPLAN Notices*, volume 48, pages 151–166. ACM, 2013.

[30] K. Sadalkar, R. Mohandas, and A. R. Pais. Model based hybrid approach to prevent sql injection attacks in php. In *Security Aspects in Information Technology*, pages 3–15. Springer, 2011.

[31] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 513–528. IEEE, 2010.

[32] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498. ACM, 2013.

[33] K. Sen, G. Necula, L. Gong, and W. Choi. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 842–853. ACM, 2015.

[34] Y. Takata, M. Akiyama, T. Yagi, T. Hariu, and S. Goto. Minespider: Extracting urls from environment-dependent drive-by download attacks. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 2, pages 444–449. IEEE, 2015.

[35] D. Y. Wang, S. Savage, and G. M. Voelker. Cloak and dagger: dynamics of web search cloaking. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 477–490. ACM, 2011.

[36] J. Wilhelm and T.-c. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *International Workshop on Recent Advances in Intrusion Detection*, pages 219–235. Springer, 2007.

[37] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee. Understanding malvertising through ad-injecting browser extensions. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1286–1295. International World Wide Web Conferences Steering Committee, 2015.

[38] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the 20th international conference on World wide web*, pages 805–814. ACM, 2011.

[39] Y. Zhou and D. Evans. Understanding and monitoring embedded web scripts. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 850–865. IEEE, 2015.