JS*: Detection and Classification of Malicious JavaScript via Attack Behavior Modelling

Yinxing Xue[†] Junjie Wang[†] Yang Liu[†] Hao Xiao[†] Jun Sun[‡] Mahin Chandramohan[†]
[†]Nanyang Technological University
[‡]Singapore University of Technology and Design

ABSTRACT

Existing malicious JavaScript (JS) detection tools and commercial anti-virus tools mostly use feature-based or signature-based approaches to detect JS malware. These tools are weak in resistance to obfuscation and JS malware variants, not mentioning about providing detailed information of attack behaviors. Such limitations root in the incapability of capturing attack behaviors in these approches. In this paper, we propose to use Deterministic Finite Automaton (DFA) to abstract and summarize common behaviors of malicious JS of the same attack type. We propose an automatic behavior learning framework, named JS*, to learn DFA from dynamic execution traces of JS malware, where we implement an effective online teacher by combining *data dependency analysis*, *defense rules* and *trace replay mechanism*. We evaluate JS* using real world data of 10000 benign and 276 malicious JS samples to cover 8 most-infectious attack types. The results demonstrate the scalability and effectiveness of our approach in the malware detection and classification, compared with commercial anti-virus tools. We also show how to use our DFAs to detect variants and new attacks.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Invasive software; D.2.5 [**Testing and Debugging**]: Tracing

Keywords

malware detection, malicious JavaScript, L*, behavior modelling

1. INTRODUCTION

In the prevalence of Rich Internet Applications [4], more and more business logics and rendering tasks are migrated from servers to clients, and such migrated logics or tasks are usually implemented in JavaScript (JS). According to Microsoft's recent security report (Fig. 81 in [13]), the prevalence of JS with associated HTML leads to the largest number of malware detected by Microsoft in the first half of 2013, e.g., the infectious *Blackhole* exploits.

Signature-based approaches are adopted by most anti-virus software, which generate a hash value or fingerprint for a malicious sample. Although these approaches can efficiently detect known malware, they fail to detect obfuscated variants with different hash values or fingerprints [52]. Additionally, from the feedback in using 56 anti-virus tools on VirusTotal [11], these tools seldom agree on the family name of detected JS malware, not to mention giving details on behaviors of the detected malware.

In parallel with signature-based approaches, there are two lines of academic approaches. For scalability and performance reasons, the first line of works mainly adopts machine learning techniques to capture malicious characteristics of malware. These approaches use syntax information (e.g., Abstract Syntax Trees [27][34]) or dynamic information (e.g., JS API [26][42]) as the predicative features for classifying malicious and benign scripts. The limitations of these approaches are two-fold. Firstly, new features can easily fail these approaches. Thus, they often require hundreds or thousands of samples for the classifiers to attain a good accuracy (like JSand [26]). Secondly, they can neither be used to classify attack types, nor can they identify new attacks from emerging malware.

The second line of approaches detects abnormal JS behaviors using dynamic analysis by honey clients or sandbox. Suspicious JS code is executed and compared with established profiles of normal JS—any inconsistency implies maliciousness. Various tools are presented to detect heap-spray attacks [29][41], worms [38][22] and other types [47]. A drawback of these tools is that they are designed for specific JS attack types, not for general JS malware detection. The detection also takes longer time than machine learning methods due to the dynamic execution. Additionally, the existing approach using defence policies, e.g., [31], can detect but fail to classify the detected malware.

Besides detection, none of aforementioned tools from security domain focuses on attack analysis based on behavior modelling. Knowing attack behavior details helps understand essences of various attacks. To further classify attacks, we treat detection of JS malware as a *behavior moodelling* problem—abstracting and summarizing common behaviors of various JS malware of the same attack type. The key idea is to effectively represent variants of the same attack type as an abstract behavior model, which could potentially provide the semantics of attack behaviors like environment detection, vulnerability exploitation and malicious payload execution. Given a suspicious JS sample, the execution of this sample produces a concrete execution trace. Checking if this trace is accepted by the inferred abstract behavior model of each attack type indicates if this trace belongs to this certain type of attack.

The challenges in JS behavior modelling stem from two aspects: (1) JS is a dynamic language with runtime evaluation and dynamic typing, which rule out the static methods. To effectively analyse its behavior requires a compact representation of JS dynamic executions and a powerful framework for dynamic analysis. (2) JS malware families are not always categorized according to their attack behaviors, sometimes according to the mode of code injection

ISSTA'15 , July 12–17, 2015, Baltimore, MD, USA
Copyright 2015 ACM 978-1-4503-3620-8 ...$15.00.

(e.g., *Trojan.js.iframe.** family launches various types of attacks). A meaningful classification according to attack types is important, especially for identifying new attacks. Such classification usually requires human expertise, and is error-prone. Thus, an automated way is needed to learn attack behaviors and do the classification.

In this study, to overcome the first challenge, we propose to analyze JS program behaviors by focusing on *browser-level system calls*. As system calls or actions are the interactions of a program (i.e., the web page in this study) with its environment (i.e., the browser in this study), it is effective to model program behaviors based on system calls or actions [21][35]. In this work, we propose to use Deterministic Finite Automaton (DFA) of the systems calls to model attack behaviors, which is inspired by the work on program behavior modelling [16]. The language accepted by the DFA includes all possible malicious executions of this type of attack, which captures the variants of the same attack.

To address the second challenge, we propose a dynamic analysis framework, named JS*, to automatically learn a DFA for each JS attack type. First, given a set of (both benign and malicious) execution traces of malicious JS, we perform a preprocessing to simplify these traces by removing security-irrelevant system calls, and the simplified (abstracted) traces are called action sequences. Second, we develop an online learning algorithm based on L* algorithm [17], which uses dynamic execution sequences to learn a DFA of the JS malware. Lastly, the inferred DFA serves as an abstract behavior model to identify variants of the modelled attack type, which can be used for both malware detection and classification. Our main contributions are as follows:

1. We treat detection and classification of malicious JS as a behaviour modelling problem. We propose to use DFA to model distinct attack types rather than behaviors of various malware families. Our approach is effective in detecting existing attack types and identifying new variants.
2. We propose JS* as a dynamic approach to automatically learn an accurate DFA from dozens of malicious samples on the fly. Rather than using a large number of training samples as in off-line learning, we combine *data dependency analysis*, *defense rules* and JS *replay mechanism* to implement an online teacher to efficiently answer if a given trace is malicious.
3. Based on real-world JS malware, we study 8 major attack types involving 120 malicious samples and learn their attack models. We test JS* with 10156 real-world JS samples, the results on which show that JS* does not only outperform existing commercial solutions in malware detection, but can also detect malware variants and even new attacks. We publish the samples and learned DFAs for public review [9].

2. JS ATTACK BEHAVIOR MODELLING

The prevalence of JS has attracted attackers to employ it for their malicious intentions. Malicious JS takes advantages of vulnerabilities or weakness of the client side with the aim to execute arbitrary instructions on the client's machine. According to Kaspersky Security Bulletin [8], attacks targeting at JRE, Adobe Reader, browsers, Adobe Flash account more than 95% of all recent attacks launched by JS code. Following this trend, this study focuses on 8 infectious and hazardous JS attacks [48], i.e., attack targeting browser vulnerabilities *(Type I)*, browser hijacking attack *(Type II)*, attack targeting Adobe Flash *(Type III)*, attack targeting JRE *(Type IV)*, attack based on multimedia (e.g., images, videos) *(Type V)*, attack targeting Adobe PDF reader *(Type VI)*, malicious redirecting attack *(Type VII)* and attack based on Web attack toolkits *(Type VIII)*. This

```
<a href='javascript:
var file=Components.classes["@mozilla.org/file/local;1"].
       createInstance(Components.interfaces.nsILocalFile);
var path = "/usr/bin/gnome-calculator";
file.initWithPath(path);
var proc=Components.classes["@mozilla.org/process/util;1"].
         createInstance(Components.interfaces.nsIProcess);
proc.init(file);
proc.run(true,[path],1);
'></a>
```

Figure 1: The JS code of a *web-based* attack

System calls sc_i**:** $(I\|M\|N_p\|S_p\|T_r)$	**Actions** a_i
nsIIOService2 | newURI | 3 | data:text/html;base64,PHN...; | void	$\Rightarrow a$
nsIURI | scheme | 0 | | void	$\Rightarrow b$
nsIPrefBranch | getComplexValue | 2 | intl.ellipsis;object; | void	$\Rightarrow n_1$
nsIPrefLocalizedString | data | 0 | | void	$\Rightarrow n_2$
nsIPrefBranch | getBoolPref | 1 | devtools.inspector.enabled; | void	$\Rightarrow c$
nsIPrefBranch | getCharPref | 1 | preview.enable; | void	$\Rightarrow c$
nsIIOService | newURI | 3 | data:text/html;base64,PHN...; | void	$\Rightarrow a$
nsIURI | scheme | 0 | | void	$\Rightarrow b$
nsILocalFile | initWithPath | 1 | /usr/bin/gnome-calculator; | void	$\Rightarrow d$
nsIProcess | init | 1 | object; | void	$\Rightarrow e$
nsIProcess | run | 3 | true;object;1; | void	$\Rightarrow f$
nsISecureBrowserUI | init | 1 | object; | void	$\Rightarrow n_3$

Figure 2: A concrete execution trace and the common actions $\{a, b, c, d, e, f\}$ **in the malicious traces of the attack in Fig. 1**

categorization includes most of commonly seen JS attacks, e.g., in the recent list of 500 malicious samples reported by WEB INSPECTOR, we find that 411 (82.2%) fall into the above 8 types of attacks.

In dynamic approaches for security analysis of various attack types, system calls are normally used to model the malicious behaviors [21][28]. The rationale is that malware triggering or payloads are often resource oriented activities, e.g., creating a process. In this work, we use FireFox as the targeted browser. In FireFox, *browser-level system calls* are system calls to the XPCOM [10] layer of Firefox (see Section 6.1). To better capture the malware behavior involving the interactions with XPCOM components, we use the lower level XPCOM method calls in this work rather than method calls of JS APIs. Here, we formally define a system call as a tuple $sc = (I, M, N_p, S_p, T_r)$, where I is the interface name of sc, M is the method name, N_p is the number of parameters, S_p is the list of arguments, and T_r is the return type. A JS execution trace is defined as a sequence of system calls $\pi = \langle sc_1, sc_2, \ldots, sc_n \rangle$, occurring in a chronological order.

EXAMPLE 1. *We list the JS of an attack in Fig. 1. Due to the vulnerability of CoolPreviews (Mozilla Firefox Extension) [5], via a link pointing to a data URI which embeds the Cross-Site scripting payload, the malicious page can inject exploiting code that is rendered and executed in the Chrome privileged zone*[1]*. Although a calculator application is used in Fig. 1, arbitrary code can be executed. Fig. 2 shows the related system calls in a malicious trace of the attack in Fig. 1. Each row represents a security relevant system call (see Section 4), where the five elements inside a system call are splitted by "$|$". The trace shown on the right column only contains 12 security relevant system calls after filtering out the security irrelevant ones from the original hundreds of system calls.*

For end-host malware behavior modeling, existing studies use two types of *behaviour atoms* [21], i.e., system call and action. The types of models include 4 basic structures (a n-gram of, a sequence of, a bag of, and a tuple of atoms), or even any combination of the basic structures [21].

Kolbitsch *et al.* [35] pinpointed that sequences of system calls are not suitable for attack behaviour modelling, as a new variant

[1]The Chrome privilege grants the JS code the permission to do everything in browser, which is similar to the root permission in OS. By default, JS is not allowed to create a file or a thread outside the sandbox with Chrome privilege.

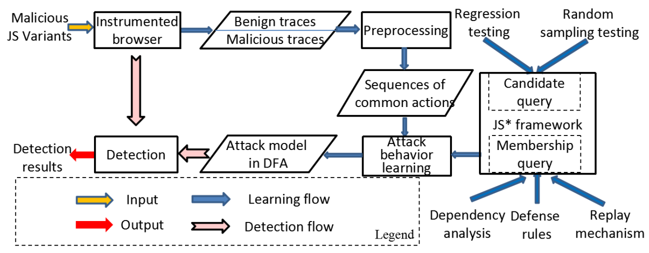


Figure 3: The general work-flow of our approach JS* to modelling the behaviors of an attack type

can be easily crafted by reordering of the code to achieve the same goals. On contrary, bags of or tuples of atoms totally ignore orders or dependencies among atoms. Thus, Kolbitsch proposed to model program behaviour in a *behaviour graph*, which is essentially a data dependency graph of system calls at runtime. In behaviour graph, two continuous system calls with no control or data dependency (no source-sink data flow) are considered isomorphic. Thus, the rationale is two-fold: system call sequence is not resilient to malware variants with reordered system calls, while bag of system calls may be too relaxed and introduce false positive cases. Behaviour graph essentially models an attack using a bag of system calls together with their data dependency. As there exist many equivalent permutations of system call sequence for system calls in the same control block, behavior graph provides an effective data dependency model among system calls. However, graph based matching of behavior graphs [35] can be computationally costly for malware detection.

Note that this issue of JS behaviour modelling has not been investigated in depth previously. Similar to using finite state automata (FSA) in modelling normal behaviour via Linux system calls [44], we propose to use DFA to model different JS attack types based on *browser-level actions*. Different from [44], our contribution is to implement an on-line teacher to perform active learning.

3. APPROACH OVERVIEW

The workflow of our approach shown in Fig. 3 contains several steps. Firstly, given the variants of an certain attack, we get both of benign and malicious execution traces by running these variants in the instrumented browser. Traces from malware variants of the same attack, not from benignware or various attacks, are used for training, as JS* targets at one type of attack each time when applied. Traces from general benignware may be irrelevant to the modelled attack, or even introduce noise for model learning. With the training traces from malware variants, the preprocessing step filters out security-irrelevant system calls and extracts actions from traces. Here, what we refer to as *actions* are high-level operations (e.g., creating a process) that can be implemented by different concrete system calls to achieve a meaningful goal [19][21]. Subsequently, action sequences converted from execution traces serve as the training set for the JS* learning (see Section 4).

In this study, the expected attack model is represented in form of a DFA $\mathcal{D}$ with a fixed alphabet Σ (i.e., common actions extracted from malicious training traces). To infer $\mathcal{D}$, the JS* learning module interacts with the polynomial-time JS* teacher via asking membership and candidate queries to make the observation table closed and consistent (see Section 5). The teacher answers membership queries effectively based on a combination of defense rules, data dependency analysis and replay mechanism (see Section 5.1). By regression testing and random sampling techniques, JS* teacher can answer candidate queries efficiently (see Section 5.2).

Finally, the learned DFA serves as an abstract attack model, which can be used to identify the attack type of the captured traces from suspicious variants. However, there may exist some action sequences that are non-deterministic, i.e., the same action sequence may be malicious or benign, depending on its arguments. In Section 5.3, we explain the inferred DFA and present a refinement for the DFA if a non-deterministic action sequence is found.

Algorithm 1: extractCommonActions

```
input  : $S_\pi = \{\pi_1, \pi_2, ...\pi_n\}$, a set of malicious system call
         traces
output: $M$, the map whose key is an action and the value is
         the set of system calls abstracted by this action.
         Initially empty.
output: $\Sigma$, the set of common actions relevant to the attack
1  foreach trace $\pi \in S_\pi$ do
2      foreach system call $sc \in \pi$ do
3          if $\exists\, a_k \in M.keys \bullet IsSameAction(a_k, sc)$ then
4              $M.get(a_k).add(sc)$ ;
5          else
6              create a new action $a_n$ according to $sc$ ;
7              $M \leftarrow M \cup \{(a_n, sc)\}$ ;
8  $\Sigma \leftarrow GetAction(\pi_1, M)$ ;
9  foreach trace $\pi \in \{\pi_2, \ldots, \pi_n\}$ do
10     $\Sigma \leftarrow \Sigma \cap GetAction(\pi, M)$ ;
```

4. TRACE PREPROCESSING

We start our approach by capturing browser-level system call traces during loading and rendering a web page through instrumenting the browser (see Section 6.1 for implementation details). Each recorded trace contains thousands of system calls, most of which are irrelevant to browser security. To precisely and concisely model attack, removing security-irrelevant system calls and mapping the rest similar ones into high-level actions prelude further analysis.

Irrelevant System Calls Filtering. To the best of our knowledge, there is no official classification of XPCOM system calls in terms of security. In this work, we record all system calls generated during the process of launching the browser, loading and rendering a totally blank web page. We consider system calls to these 667 (out of the 1948 interfaces in XPCOM [10]) basic and very common interfaces as no security risks, i.e., *security-irrelevant system calls*, e.g., system call nsIDOMWindow.GetscreenY that gets the Y coordinate of a window. Most of these 667 interfaces are related to browser initialization and configuration. In our study, system calls uncommonly appearing in normal browser launching are *security-relevant system calls*, e.g., system call nsIProcess.run that executes a process. Finally, relevant system calls are listed in our website[9].

Action Abstraction. Some system calls provide similar or identical functionalities, e.g., in Fig. 2, nsIPrefBranch.getBoolPref and nsIPrefBranch.getIntPref get the Bool and Int type preference data; nsIIOService1.newURI and nsIIOService2.newURI both construct a new URI, respectively. Thus, we abstract system calls with similar functionalities as the same type of *action* for the purpose of variants detection. We name the actions alphabetically starting from ‘a’ in this work. After the abstraction, malicious traces based on system calls become abstracted action sequences. Due to multitasking of the browser, each execution of the same JS code may produce different action sequences but with the identical attacking route. Intersecting these action sequences extracts common actions, making visible common behaviours of malicious traces.

Algorithm 1 illustrates how system calls are mapped into actions and then how common actions are extracted from the given set of malicious traces S_π. First, the map M is built from lines 1 to 7. At line 3, each system call sc in π from S_π is compared with each key a_k in M by calling $IsSameAction$ to check the functionality

Malicious Action Sequences S_π:	**Benign Action Sequences S'_π:**
$\pi_{m1} : \langle c.a.b.d.e.f \rangle$	$\pi_{b1} : \langle c.b.d.e.f \rangle$
$\pi_{m2} : \langle a.c.b.d.e.f \rangle$	$\pi_{b2} : \langle a.c.d.e.f \rangle$
$\pi_{m3} : \langle a.b.c.d.e.f \rangle$	$\pi_{b3} : \langle a.b.c.e.f \rangle$
$\pi_{m4} : \langle a.b.d.c.e.f \rangle$	$\pi_{b4} : \langle a.b.d.c.f \rangle$
$\pi_{m5} : \langle a.b.d.e.c.f \rangle$	$\pi_{b5} : \langle a.b.d.e.c \rangle$
$\pi_{m6} : \langle a.b.d.e.f.c \rangle$	$\pi_{b6} : \langle c.b.a.d.e.f \rangle$
$\pi_{m7} : \langle c.a.c.b.c.d.c.e.c.f \rangle$	$\pi_{b7} : \langle c.a.b.e.d.f \rangle$
$\pi_{m8} : \langle a.b.c.c.a.b.d.e.f \rangle$	$\pi_{b8} : \langle c.a.b.d.f.e \rangle$

Figure 4: Representing traces as sequences of common actions (π_{m8} is the counterpart of the trace in Fig. 2)

similarity, at Line 3. If the fully qualified name of sc is similar to the fully qualified name of any system call in set $M.get(a_k)$, sc belongs to $M.get(a_k)$. Being similar is satisfied if string similarity according to Levenshtein Distance [30] is above a threshold, 80% in our study [37]. We also refine the results of this step via the manual check, considering the limited size of system calls involved in each type of attack. Method $M.get(a_k)$ returns those system calls abstracted by a_k. If *IsSameAction* returns true, i.e., sc can be abstracted by action a_k, sc is added into $M.get(a_k)$ (line 4). Otherwise, a new action a_n is created according to next available alphabet (ASCII letter in our implementation) at line 6; and the pair (a_n, sc) is added to M at line 7. Practically, action abstraction step is manually verified, as the size of alphabet is small (see section 6).

After M is built up, method $GetAction(\pi_1, M)$ at line 8 collects the actions in π_1 and assigns to Σ. From lines 8 to 10, all action sequences are iteratively used to build the set of common actions Σ—actions appearing in all traces in S_π.

Note that extracting common actions is only conducted on malicious traces. Only after that, the available benign traces are represented as sequences of common actions. Suppose we collected 8 malicious and 8 benign concrete system call traces, we apply action abstraction and intersection on these 8 malicious traces to build the set of common actions, i.e., $\{a, b, c, d, e, f\}$ in Fig. 2. Finally, in Fig. 4 we represent these 16 traces in sequences of common actions, which serve as the input for the JS* learning approach.

5. JS* LEARNING FRAMEWORK

This section is devoted to the explanation of the proposed JS* learning framework. Our approach is based on the L* algorithm, which learns a DFA from a set of strings [17]. For string learning, L* contains a teacher to answer membership queries and candidate queries via substring or regular expression matching. In our study, we combine domain knowledge (e.g., defence rules), program analysis (e.g., data dependency), and other techniques (e.g., replay mechanism, random sampling) to implement an effective teacher to answer these two queries (see Sections 5.1 and 5.2).

DEFINITION 1. *A deterministic finite automaton (DFA) is a 5-tuple $\mathcal{D} = (S, \Sigma, \delta, \lambda, F)$, where S is a finite set of states; Σ is the finite set of input events—so called alphabets; δ is a transition function defined as $\delta : S \times \Sigma \to S$; $\lambda \in S$ is an initial state; $F \subseteq S$ is a set of accept states.*

First, we formally define DFA as above. In this study, we assume that an attack model m is in the form of a DFA $\mathcal{D}$, which has a fixed alphabet Σ and a language $\mathcal{L} \subseteq \Sigma^*$. The L* algorithm learns a DFA $\mathcal{C}$ with a set of states S with the minimal size accepting the same language $\mathcal{L}$ of $\mathcal{D}$. During the learning process, a Minimal Adequate Teacher (teacher for short) is entailed to answer the two types of queries: membership queries and candidate queries. The former asks whether a given *trace* (alphabetical sequence) should be accepted by $\mathcal{D}$, while the latter asks whether the given DFA $\mathcal{C}$ is equivalent to $\mathcal{D}$, i.e., accepting the same language. Thus, when the learning stops, the inferred $\mathcal{C}$ is equivalent to the expected $\mathcal{D}$.

Algorithm 2: membershipQuery

```
input  : $\pi_a$, a given action sequence, which cannot be null
output: $true$ or $false$, if $\pi_a$ should be accepted by the
        expected DFA.
1 if $\exists \pi_i \in S_\pi \bullet \pi_i$ is a prefix of $\pi_a$ then
2     return $true$;
3 if $\exists \pi'_i \in S'_\pi \bullet \pi_a == \pi'_i$ then
4     return $false$;
5 if $isEqualPermutation(\pi_a, S_\pi)$ then
6     return $true$;
7 if $\pi_a$ violates any defense rule then
8     return $true$;
9 return $Replay(\pi_a)$ ;
```

During learning, to store the results of membership queries, L* algorithm maintains an *observation table* (P, Q, T), where $P \subseteq \Sigma^*$ is a set of prefixes; $Q \subseteq \Sigma^*$ is a set of suffixes; and $T : (P \cup P \cdot \Sigma) \times Q \mapsto \{0, 1\}$ is a mapping function such that if $p \cdot q$ is a trace accepted by $\mathcal{D}$, then $T(p, q) = 1$; otherwise (i.e., $p \cdot q$ is a trace rejected by $\mathcal{D}$), then $T(p, q) = 0$, where $p \in (P \cup P \cdot \Sigma)$ and $q \in Q$. The observation table categorizes the given traces according to Myhill-Nerode Congruence [32].

DEFINITION 2. *(Myhill-Nerode Congruence). For any two traces $tr, tr' \in \Sigma^*$, they are* equivalent*, denoted by $tr \equiv tr'$, if $tr \cdot q$ is accepted by $\mathcal{D}$ iff $tr' \cdot q$ is accepted by $\mathcal{D}$, for all $q \in \Sigma^*$. We denote such equivalency by $tr = [tr']_r$ and $tr' = [tr]_r$, considering tr and tr' are the* representing traces *of each other with respect to the language $\mathcal{L}$ accepted by $\mathcal{D}$.*

In the L* learning process, the observation table (P, Q, T) should be *closed* and *consistent* with the membership queries to reach a candidate DFA. Being closed is satisfied when for all $p \in P$ and $q \in \Sigma$, there exists $p' \in P$ such that $p \cdot \langle q \rangle \equiv p'$. Meanwhile, being consistent is satisfied when for any two trace $p, p' \in P$ such that $p \equiv p'$, and also $(p \cdot \langle q \rangle) \equiv (p' \cdot \langle q \rangle)$ for all $q \in \Sigma$. When (P, Q, T) is closed and consistent, a candidate DFA $\mathcal{C}$ is constructed and denoted as $\mathcal{C} = (S_C, \Sigma_C, \delta_C, s^0_C, F_C)$ such that $S_C = P$, $\Sigma_C = \Sigma$, $\delta_C(p, q) = [p \cdot q]_r$ for $p \in P$ and $q \in \Sigma$, $s^0_C = \{\lambda\}$, and $F_C = \{p \in P \mid T(p, \lambda) = 1\}$. Subsequently, the L* uses $\mathcal{C}$ as the input DFA for a candidate query.

The teacher answers the candidate query by checking the equivalence of $\mathcal{C}$ and the black box DFA $\mathcal{D}$. If the answer is true, $\mathcal{C}$ is exactly $\mathcal{D}$ that needs to be modeled, and the learning terminates. Otherwise, the teacher provides a counterexample for the L* to identify a *witness suffix*, which is a trace that—when appended to the two other traces—provides enough evidence for these two traces to be classified into different equivalence classes under the Myhill-Nerode Congruence. After such witness suffix is identified, it will be used by the L* to refine the candidate DFA $\mathcal{C}$ until $\mathcal{C}$ is equivalent to $\mathcal{D}$. Details on the L* algorithm can be found from [17]. It is proved that for a regular deterministic language the L* only needs asking at most $n - 1$ candidate queries and $O(|\Sigma| n^2 + n \log m)$ membership queries [17]. If the teacher answers membership and candidate queries in polynomial time, the whole learning process is completed in polynomial time.

5.1 Membership Query

An ideal implementation of the on-line teacher for membership queries should be able to correctly answer if the system call sequence $\pi_a \in \Sigma^*$ should be accepted (i.e., whether π_a represents an attack of the modelled type) in polynomial time. Practically, it is infeasible to extract all possible traces for a JS program and judge their maliciousness. Here, we propose a feasible and efficient way to utilize defence rules, data dependency analysis, and

Algorithm 3: isEqualPermutation

```
input : $S_\pi = \{\pi_1, \pi_2, ...\pi_n\}$, a set of malicious action
        sequences
input : $\pi_a$, a given action sequence
output: $true$ or $false$, if $\pi_a$ is an equivalent permutation
1 $S_{e\pi} \leftarrow \emptyset$ ;
2 foreach sequence $\pi_i \in S_\pi$ do
3     $S_{dc} = getDependencyClosure(\pi_i)$ ;
4     $S_{e\pi} \leftarrow S_{e\pi} \cup getPermutations(\pi_i, S_{dc})$ ;
5 if $\pi_a \in S_{e\pi}$ then
6     return $true$;
7 return $false$;
```

also JS replay mechanism to answer membership queries on the fly, as elaborated below.

5.1.1 *Browser Defense Rule*

Defense rules refer to security policies commonly used by the mainstream browsers to detect malicious attacks. Violation of such rules indicates possible security risks in existing studies [45]. Mozilla mainly adopts permission relevant policies: the *same origin policy* (used in [31][33]), the *Configurable Security Policies (CAPS)* as Mozilla built-in zone-based rules [6], and the *signed-script policy* (used in [31][53]). In our study, these defense rules are applied to the URL source, information and permission of executed JS code. By combining these rules in the teacher, we can test a trace—a violation of any defense rule indicates that the tested trace is malicious.

Note that defense rules are mainly used to check the trace that are tested by the JS* on the fly. Besides, using defense rules can only indicate whether a trace is benign or not [31], but fail to model the attack behaviors.

5.1.2 *Data Dependency Analysis*

Given the existing training traces, data dependency analysis is adopted to find equivalent permutations (EPs) of an action sequence. The assumption that the order of two mutually independent system calls (or actions) does not matter is reasonable [21][35]. Thus, we adopt this assumption and infer EPs via data dependency analysis. For example, given malicious sequence $\pi_1 = \langle a_1, a_2, a_3 \rangle$, where a_3 has data dependency[2] on a_1 and a_2, denoted as $\{a_3 \leftarrow (a_1, a_2)\}$, we can infer $\pi_2 = \langle a_2, a_1, a_3 \rangle$ is also malicious. So π_2 and π_1 are EPs.

Given a training set of malicious sequences S_π and an unknown system call sequence π_a, Algorithm 3 shows how to test if π_a is an EP according to the sequences in S_π. The basic idea is to get the data dependency closures among actions inside a sequence π_i by invoking method $getDependencyClosure(\pi_i)$. In two steps, this method builds the dependency closure[3]. Given π_i, JS* gets all direct dependencies among system calls inside this trace, from which actions are abstracted. If two system calls have a direct data dependency (see Section 6.1), the two relevant abstracted actions have a direct data dependency. Second, this method propagates the direct data dependency relationship among actions into a closed transitive indirect data dependency. In the running example, this method returns two closures $\{b \leftarrow a\}$ and $\{(e, f) \leftarrow d\}$. As c is independent, actually, π_{m1} to π_{m6} (and other similar sequences only with different c positions) are all EPs.

By $getPermutations$, the mutually independent relation between actions (e.g., a_1 and a_2 in the above case of π_1, c and other actions in the case of π_{m1}) infers the equivalent permutations. All EPs should not equal to an existing benign sequence, and then be stored into a set for comparison with π_a. A right match indicates an EP. Practically, the calculation of $S_{e\pi}$ from lines 1 to 4 in Algorithm 3 is pre-built once as preprocessing. Due to the limited length of action sequence and the small size of training sequences, $getPermutations$ is scalable in reality, e.g., for the partial DFA in Fig. 8, there is the dependency closures: $\{n \leftarrow m \leftarrow l \leftarrow k \leftarrow j \leftarrow i \leftarrow (g, h) \leftarrow f \leftarrow (c, d, e) \leftarrow b \leftarrow a\}$. Thus, (g, h) and (c, d, e) are two sets of independent actions that lead to 12 EPs—multiplied by 2 for (g, h) and 6 for (c, d, e).

JEIS statement	Action
1. nsIPrefBranch.getBoolPref("devtools.inspector.enabled");	$\Rightarrow c$
2. var nsIIOService2=Components.classes["@mozilla.org/network/io-service;1"].getService(Components.interfaces.nsIIOService);	N.A.
3. var nsIURI=nsIIOService2.newURI("data:text/html;base64, PHNjcmlwd...",null,null);	$\Rightarrow a$
4. nsIURI.scheme();	$\Rightarrow b$
5. var nsILocalFile=Components.classes["@mozilla.org/file/local;1"].createInstance(Components.interfaces.nsILocalFile);	N.A.
6. nsILocalFile.initWithPath("/usr/bin/gnome-calculator");	$\Rightarrow d$
7. var nsIProcess==Components.classes["@mozilla.org/process/util;1"].createInstance(Components.interfaces.nsIProcess);	N.A.
8. nsIProcess.init(nsILocalFile);	$\Rightarrow e$
9. nsIProcess.run(true,['/usr/bin/gnome-calculator'],1);	$\Rightarrow f$

Figure 5: The JEIS statements reverse-engineered from action sequence $\pi_{m1} : \langle c.a.b.d.e.f \rangle$

Data dependency analysis is used to check the benignity of traces on the fly in our teacher implementation. Based on dependency closures inferred from malicious sequences, JS* identifies benign sequences not holding these closures, and find malicious EPs. Sequences $\langle a.b.d.e.f \rangle$ (removing prefix c from π_{m1}) and $\langle c.a.b.d.f.e \rangle$ (partial reordering of π_{m1}), which satisfy dependency closures but are unknown for benignity, will be tested with replay mechanism as presented below.

5.1.3 *Trace Replay*

Replay mechanism is implemented to dynamically test maliciousness of an inquired trace during on-line learning. It is implemented using JS's Equivalent Intermediate Script (JEIS), which is the intermediate code rather than the source-code. The basic idea is to manually craft a JEIS with Chrome privileges, according to a given action sequence. We do not craft JEIS from scratch, but create a JEIS by adding, deleting or reordering JEIS statements from the reverse-engineered JEIS of existing training traces.

As explained at the beginning of this section, a new sequence usually has a prefix or suffix added to or deducted from a previous sequence that has been tested in a membership query. For example, during learning, a membership query checks sequence $\langle a.b.d.e.f \rangle$ (removing prefix c from π_{m1}). To replay $\langle a.b.d.e.f \rangle$, we get the reverse-engineered JEIS of π_{m1}, then remove the intermediary script relevant to c, and finally craft the expected JEIS that contains statements **2-9** in Fig. 5. Executing this JEIS in JS engine realizes replay mechanism for sequence $\langle a.b.d.e.f \rangle$. As replaying this JEIS produces the same result as that of running JEIS of π_{m1}, we claim that $\langle a.b.d.e.f \rangle$ is also malicious.

JEIS execution produces three types of outcomes. First, JEIS execution may trigger some defence rules, or some heuristic rules that detect malware in the sandbox of JS engine [31], e.g., a JEIS with system call nsIProcess.init whose origin is from an external website violates CAPS, as its interface nsIProcess cannot be executed without Chrome privilege. In case of running JEIS of π_{m1} and $\langle a.b.d.e.f \rangle$, we consider they are malicious and cause potential risks. Second, JEIS execution may cause some runtime exceptions or crashes. Such exceptions or crashes indicate that the replayed sequence is infeasible rather than malicious, e.g., according

[2]The data dependency is calculated based on the original system calls of the actions, we omit the implementation details for the interest of space.

[3]A dependency closure is a transitive relationship among actions, and each in this closure is directly or indirectly involved in data dependencies on others.

to JEIS of $\pi_{m1}\langle c.a.b.d.e.f\rangle$ in Fig. 5, we want to craft the JEIS of $\langle c.a.b.d.f.e\rangle$ that has a reversed order of e and f. Inverting e and f, and running the new crafted JEIS (i.e., the JEIS statements in the order of ⟨**1,2,3,4,5,6,7,9,8**⟩) causes an exception in the JS engine, as nsIProcess.run (statment **9**) is executed before nsIProcess.init (statment **8**). So $\langle c.a.b.d.f.e\rangle$ is infeasible, and this infeasible sequence means that the outcome is benign. Lastly, if JEIS execution fails to produce any obvious resource oriented activities and violates no rules, it is assumed benign.

5.1.4 *Membership Query Algorithm*

Given malicious action sequences S_π and benign sequences S'_π, e.g., the sequences in Fig. 4, Algorithm 2 describes the process in answering the membership query regarding to the given action sequence π_a.

First, if any $\pi_i \in S_\pi$ is a prefix of π_a (line 1), i.e., π_a equals to or starts with π_i, π_a must be malicious. If π_a equals any sequence in S'_π (line 3), π_a must be benign. Then $isEqualPermutation(\pi_a, S_\pi)$ method at line 5 defined in Algorithm 3 is called to check whether π_a is an EP, and a *true* answer means that π_a is also a malicious EP. The next step is to check if π_a violates the defense rules in Section 5.1.1 at line 7. Finally, method $Replay(\pi_a)$ at line 9 is called to concretely execute π_a using replay mechanism to verify its benignity.

In our running example, an example of satisfying the check at line 1 is action sequence $\langle c.a.b.d.e.f.a\rangle$, which is considered as malicious as it starts with an existing malicious sequence π_{m1}. An example for $isEqualPermutation(\pi_a, S_\pi)$ at line 5 is to check action sequences $\langle c.a.b.e.f.d\rangle$ and $\langle a.b.c.c.d.e.f\rangle$. According to the collected traces like π_{m8} in Fig. 2, we found that action b has data dependency on a; c is independent; meanwhile e and f have a data dependency on d. Thus, $\langle c.a.b.e.f.d\rangle$ is not an equivalent permutation of existing malicious sequences like $\langle c.a.b.d.e.f\rangle$. But $\langle a.b.c.c.d.e.f\rangle$ is malicious as it satisfies the identified data dependency, similar to π_{m3} but with only one more independent c.

At line 7, our example in Fig. 1 does not violate any rule, as all the scripts are from the same origin, and no signed-script is used. Lastly, $Replay(\pi_a)$ at line 9 is effective in running the left uncertain sequences ranging from simple ones $\langle a.b\rangle$ to those complicated ones like $\langle b.a.f.a.b.f.e.d.a.e.d.d.f\rangle$ for membership querying. Actually, $Replay(\pi_a)$ all returns *false* for these two queries, as the two replayed sequences produce no malicious results (no resource oriented activities and no permission/rule violations).

5.2 Candidate Query

During learning, an intermediate DFA $\mathcal{C}$ is inferred after multiple membership queries. To judge if $\mathcal{C}$ is equivalent to the expected DFA $\mathcal{D}$ that accurately models the attack, an efficient algorithm is required for the validity check in polynomial time. When a candidate query is evaluated, two types of counterexamples can be found on $\mathcal{C}$—false positives and false negatives. The former means that a sequence accepted by $\mathcal{C}$ should be rejected by $\mathcal{D}$, while the latter means that a sequence rejected by $\mathcal{C}$ should be accepted by $\mathcal{D}$.

Given S_π and S'_π in Fig. 4, Algorithm 4 illustrates how the teacher answers the candidate query for the given $\mathcal{C}$, based on regression testing and random sampling testing. First, at line 1, each sequence π_i from the known sequence sets S_π, S'_π and the previously tested sequence set $S_{o\pi}$ is input to $membershipQuery$ and also $\mathcal{C}.isAccepted()$. Method $membershipQuery(\pi_i)$ at line 2 returns true if π_i is accepted by $\mathcal{D}$. If $\mathcal{C}$ and $\mathcal{D}$ show different acceptance results for π_i, a counterexample π_i is found and returned. Second, a random walk function $randomWalks()$ is used to generate $S_{n\pi}$, a new set of random sequences that include both

Algorithm 4: candidateQuery

```
input : $\mathcal{C} \neq NULL$, the learned candidate DFA
input : $S_{o\pi}$, the set of old sequences that have been tested in
        previous calls of candidateQuery, initially being $\emptyset$
        before any $candidateQuery$ is called
output: $\pi_{ce}$, an counterexample sequence found
1 foreach trace $\pi_i \in (S_\pi \cup S'_\pi \cup S_{o\pi})$ do
2     if $membershipQuery(\pi_i) \neq \mathcal{C}.isAccepted(\pi_i)$ then
3         return $\pi_{ce} \leftarrow \pi_i$;
4 $S_{n\pi} \leftarrow$
  $randomWalks(\mathcal{C}, \mathcal{C}.stateSize * times, \mathcal{C}.stateSize + extraLenLmt)$ ;
5 foreach trace $\pi_i \in S_{n\pi}$ do
6     $S_{o\pi} \leftarrow S_{o\pi} \cup \{\pi_i\}$;
7     if $membershipQuery(\pi_i) \neq \mathcal{C}.isAccepted(\pi_i)$ then
8         return $\pi_{ce} \leftarrow \pi_i$;
9 return $\pi_{ce} \leftarrow NULL$ ;
```

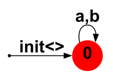
accepted and rejected ones on $\mathcal{C}$. The rationale of using random walk is that sequences on $\mathcal{C}$ cannot be enumerated due to potential loops. $randomWalks()$ at line 4 has three parameters, where $\mathcal{C}$ is the candidate DFA; $\mathcal{C}.stateSize{*}times$ denotes the number of generated sequences and $times$ is an input constant to multiply; $\mathcal{C}.stateSize{+}extraLenLmt$ is the maximum allowed length of generated sequences and $extraLenLmt$ is the extra length that can be larger than $\mathcal{C}.stateSize$. Here, $\mathcal{C}.stateSize$ denotes the size of total states in $\mathcal{C}$. Then, each sequence π_i in $S_{n\pi}$ is also given to $membershipQuery()$ and $\mathcal{C}.isAccepted()$ for acceptance check—an inconsistency indicates a counterexample. Note that any tested π_i from $S_{n\pi}$ is added to $S_{o\pi}$, which is used for regression testing in answering the next candidate query. Finally, if no counterexample is found, $NULL$ is returned and the learning process stops.

In our running example, candidate queries are asked twice. The first candidate DFA $\mathcal{C}_1$ in Fig. 6 (a) is inferred when P in the observation table only contains $\{\lambda, a, b, c, d, e, f\}$ with one state 0. For $\mathcal{C}_1$, Algorithm 4 returns a malicious sequence $\pi_{m1} : \langle c.a.b.d.e.f\rangle$ as a counterexample for further learning. Afterwards, a candidate DFA $\mathcal{C}_2$ in Fig. 6 (b) is learned. According to regression testing and random sampling testing in Algorithm 4, no counterexample for $\mathcal{C}_2$ is found—$\mathcal{C}_2$ is equivalent to the expected $\mathcal{D}$.

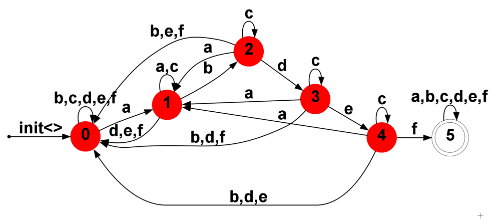
5.3 The Learned DFA and Refinement

Given the 16 sequences in Fig. 4, JS* undergoes 622 times membership queries and 2 times candidate queries (with $times{=}5$ and $extraLenLmt{=}5$ for method $randomWalks$ in Algorithm 4), and infers a DFA $\mathcal{D}$ to model the attack—or equivalently a regular language $\mathcal{L} = (c)^* \cdot a \cdot (c)^* \cdot b \cdot (c)^* \cdot d \cdot (c)^* \cdot e \cdot (c)^* \cdot f \cdot (c)^*$ over $\Sigma = \{a, b, c, d, e, f\}$. To apply this DFA to detect malicious variants of this attack, a trace from a suspicious variant is collected and preprocessed to be converted into an action sequence π over Σ. An acceptance of π on $\mathcal{D}$ suggests that π is from a malicious trace.

The sequences in S_π in Fig. 4 are all deterministic (being certainly malicious), and produced by the script with fixed arguments in Fig. 1. In practice, the same sequence of actions might be sometimes malicious and sometimes benign, depending on the arguments of the calls. For instance, if the last argument of the statement proc.run(true,[path],1) in Fig. 4 is changed from "1" to "0". Executing the new code can produce the same sequences as those old ones in Fig. 4, and all data dependencies inside these new sequences still hold. But the new sequence $\pi'_{m1} : \langle c.a.b.d.e.f\rangle$ is benign and creates no process, since inside π'_{m1} the last argument of action f is "0", which makes nsIProcess.run include zero argument from the argument list. One way to solve the problem is



(a) candidate DFA $\mathcal{C}_1$



(b) candidate DFA $\mathcal{C}_2$

Figure 6: The learned candidate DFA $\mathcal{C}_1$ and $\mathcal{C}_2$

to model system calls with different arguments as different actions, for instance, we can represent the above as: $\pi_{m1} : \langle c.a.b.d.e.f(1)\rangle$ and $\pi'_{m1} : \langle c.a.b.d.e.f(0)\rangle$. To refine the DFA for such case, we derive new actions f_m and f_b from current action f. Here, f_m refers to the action $f(\{s_1, ...s_n\})$ with the argument set $\{s_1, ...s_n\}$ that produces malicious outcome like $f(1)$. Similarly, f_b refers to the action f with arguments that produce benign outcome. Then with the new $\Sigma' = \{a, b, c, d, e, f_m, f_b\}$, JS* is applied again on the training sequences to learn a refined DFA. Thus, JS* supports refinement for nondeterministic sequences in a reactive way.

6. IMPLEMENTATION AND EVALUATION

We conduct experiments on real-world JS malware to study 8 popular attack types to evaluate JS*. We aim at answering the following 4 questions in our evaluation:

RQ1. Do our learned DFAs *correctly* and *effectively* model common and abstract behaviors for each attack type?

RQ2. Does JS* perform *efficiently* in the learning process in terms of the running time?

RQ3. Is JS* *accurate* in the malware detection, compared to other research prototypes and commercial anti-virus products?

RQ4. Are these learned DFAs *useful* in detection of emerging malicious variants and new attack by unknown malware?

RQ1 and RQ2 examine the effectiveness and efficiency of JS*. RQ3 and RQ4 are to investigate the usefulness of the inferred DFAs.

6.1 Implementation

We implement JS* based on instrumentation to Firefox kernel. There are three layers in Firefox's JS execution environment. At the upper layer, Firefox JS engine SpiderMonkey interprets JS code and invokes the related low layer libraries via XPConnect, which provides interaction between SpiderMonkey and XPCOM. As XPConnect bridges the top and bottom layer, we hook calls to methods in XPConnect interfaces as JS browser-layer system calls. By comparing the affected objects (e.g., the returned object, changed objects or arguments) of two sequential system calls sc_1 and sc_2 in terms of object type and memory address, the data dependency $\{sc_1 \leftarrow sc_2\}$ is checked. In JS*, we propagate data dependency check using data flow analysis to calculate transitive dependency relationship used in data dependency analysis (Section 5.1.2).

6.2 Data Preparation and Setup

To evaluate JS* on the eight popular types of JS attacks mentioned in Section 2, we collect 276 distinct malicious samples out of more than 1000 real-world malware[4] that originate from various sources: 40 unique samples from VXHEAVEN [3]; 66 unique ones from OPENMALWARE [2]; and we also manually collected 170 samples from the most recent list of malicious websites reported by WEB INSPECTOR [12]. We manually inspect these 276 samples to verify their maliciousness. Besides, we collect 10000 benign

[4]There is a large number of duplicated and expired malware in our collected samples.

Attack Type	$\|S\|$	$\|\Sigma\|$	#M-Q	#C-Q	Time(s)	Total Time(s)
Type I	67	29	132,084	4	1.83	945
Type II	77	11	33,585	7	0.501	634
Type III	54	11	41,459	8	4.37	902
Type IV	74	28	158,120	4	1.95	1,068
Type V	121	29	468,124	6	8.33	2,768
Type VI	50	11	34,266	4	0.551	498
Type VII	466	15	309,277	20	22.425	4,597
Type VIII	74	11	28,480	12	0.563	993

Table 1: The learning results of JS*

samples from the Alexa [1] top 100 web sites, none of which is reported as malicious by the 56 tools provided by VIRUSTOTAL[5].

Among the total 276 malicious samples, we select 120 samples ($\approx$40%), i.e., 15 for each attack type, as the training set. Before JS* learning, each sample is executed 10 times to get 10 traces, most of which differ from each other due to different browser context at the time of running. These traces are converted to similar action sequences after preprocessing. Among 8 different attack types, at least 37% (for *Type V*) to at most 60% (for *Type VII*) of traces are unique. We apply JS* separately for 8 attack types based on the traces executed from their training samples. To verify the inferred DFAs, we use the remaining 156 malicious samples, most of which are recently reported by WEB INSPECTOR, together with the 10000 benign samples as the testing data set for predication.

The experimental environment is a PC with Intel i7 2600 3.4GHz CPU and 8GB memory. The system environment is Ubuntu 12.04 and the Firefox that we instrumented in JS* is 17.0.

6.3 JS* Learning Evaluation

The statistics of the learning process as well as the inferred DFAs are listed in Table 1. $|S|$ denotes the size of the states inside the inferred DFA; $|\Sigma|$ denotes the size of the alphabetic of the DFA (the size of common actions); **#M-Q** refers to the number of called membership queries; **#C-Q** refers to the number of called candidate queries; **Time(s)** denotes the core time (in seconds) of learning process; **Total Time(s)** denotes the total time (in seconds) of learning process, including trace generation and replay.

RQ1: Correctness. We validate the correctness of these DFAs from two aspects: (1) identifying the high level semantics by checking their alphabets, and (2) interpreting the accepted path of DFAs with hints in the descriptions of the CVE used by the attack.

First, checking the comparatively small set of common actions Σ briefly tells whether common essential behaviors of the same attack type are captured and modelled. From our observation, the set of common actions (Σ) for each attack type is reasonable and relevant to the attack type. For example, *Type I* attack can be generally divided into three steps: first, putting the shell code in a predictable memory location; then triggering an exploitable crash (modelled by common actions like nsIAppStartup.trackStartupCrashEnd and others in the alphabet of *Type I*); at last, the shell code will be executed to perform the attack, invoking file operations system calls (modelled by common actions nsIFile.append, etc).

Second, we check accepted traces of *Type VI* attack DFA and identify five steps: downloading malicious pdf file, executing embedded JS to scan vulnerability, exploiting the vulnerability, executing payload, and actual sabotage. The detailed alphabet of each learned DFA and the explanations can be found in our website [9].

We observed that the eight inferred DFAs share some common parts, e.g., payloads executing. Generally, payloads include executing arbitrary command, binding shell or reversed shell using TCP, etc. In Fig. 7, we illustrate the common behaviors of binding shell

[5]To our best knowledge, JSAND is the only open service for JS malware detection, and VIRUSTOTAL is powered by updated versions of mainstream anti-virus products.

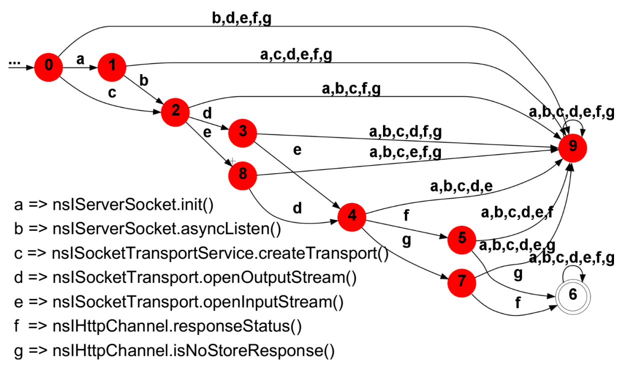

...
0
1
2
3
4
5
6
7
8
9
a
b
c
d
e
f
g
b,d,e,f,g
a,c,d,e,f,g
a,b,c,f,g
a,b,c,d,e,f,g
a,b,c,d,f,g
a,b,c,e,f,g
a,b,c,d,e
a,b,c,d,e,f
a,b,c,d,e,g
a,b,c,d,e,f,g
a => nsIServerSocket.init()
b => nsIServerSocket.asyncListen()
c => nsISocketTransportService.createTransport()
d => nsISocketTransport.openOutputStream()
e => nsISocketTransport.openInputStream()
f => nsIHttpChannel.responseStatus()
g => nsIHttpChannel.isNoStoreResponse()

Figure 7: The DFA of binding shell using TCP, which serves a partial DFA of all attack types

or reversed shell using TCP, after the exploit for each corresponding attack is done.

The unique parts of an inferred DFA model the essential attack behaviors of the corresponding attack type. In Fig. 8, we show a fraction of *Type I* DFA. This fraction exploits the CVE-2013-1710 vulnerability. Specifically, this attack invokes system call crypto.generateCRMFRequest to enable remote attackers to execute arbitrary JavaScript code or conduct cross-site scripting (XSS) attacks via vectors related to Certificate Request Message Format (CRMF) request generation. The original trace contains 1,236 system calls and has 12 equivalent permutations with the same set of actions. However, with our JS* learning approach, the DFA in Fig. 8 contains only 14 actions and 19 states. Thus, our inferred attack behavior models in form of DFA are concise yet accurate, without loss of the essence of attack.

A notable case is that the DFA of malicious redirecting attack *(Type VII)* has 466 states. A possible explanation is that samples of *Type VII* attack used for learning are less similar than samples of other attack types. Thus, 15 dissimilar samples infer a DFA with a small alphabetical size but a large size of states, which does not necessarily means a bad modelling result. On contrary, it suggests that there exist many traces from this DFA to be accepted—more possible variants of this attack type. Such explanation is backed up by the fact that malicious redirecting attacks are simple with a small set of common behaviours ($|\Sigma|$ =15), but flexible with possibly enormous variants: 986 out of totally 1300 malicious JS samples provided by VXHEAVEN are drive-by-download attacks that generally relate to *Type VII* attack.

Effectiveness. Besides manual observation, we empirically validate the usefulness of the DFAs by conducting predication. On the testing set of 10000 benign and 156 malicious samples, each sample is executed 10 times to get different traces. If any trace is accepted by one of eight learned DFAs, the corresponding sample is detected as one variant of the attack that is modelled by the matched DFA. Totally, JS* correctly detects 149 out of 156 (95.51%) malicious samples and 9957 out of 10000 (99.57%) benign samples. The distributions of 7 FN cases (4.49%) and 54 FP cases involving 43 distinct benign samples (0.43%) are listed in column **JS* FN** and **JS* FP** of Table 2[6], respectively. Among 43 distinct benign samples, 11 (54-43) are falsely accepted by two DFAs, e.g., 2 benign samples are accepted by DFAs of both *Type VII* and *VIII*, which share commonality — toolkits based attacks utilize abnormal redirection iteratively to evade detection. Thus, JS* attains low overall FN ($\approx$5%) and FP ($\approx$0.5%) rate for all 8 attack types.

[6]The reason to put Total at the first row is that there is no type-based detection in Jsand or the 56 tools on VirusTotal. We separately list the 8 type-specific rows to show how attack types affect detection results.

Type	JS*			JSAND		V.T.
	FP	FN	T(S)	FN	T(S)	
Total	43* /10000	7 /156	1.36	95/156	4.8	21
Type I	4/10000	0/15	1.31	12/15	3.3	19
Type II	9/10000	1/15	1.34	12/15	5.9	21
Type III	3/10000	0/18	1.50	13/18	4.2	17
Type IV	7/10000	1/15	1.32	9/15	3.3	20
Type V	11/10000	2/20	1.33	12/20	3.4	21
Type VI	8/10000	0/23	1.32	2/23	8.3	34
Type VII	3/10000	1/23	1.26	18/23	3.4	15
Type VIII	9/10000	2/27	1.47	17/27	5.3	21

Table 2: The predication results of JS*, JSAND and 56 tools provided by VIRUSTOTAL

RQ2: Performance. The results reported in column **Time(s)** of Table 1 show that our approach is highly scalable in the core learning process. The required learning time is generally proportional to the state number and the alphabetical size of the learned DFA. Column **Total Time(s)** includes the time used for trace generation and replay, which is the major overheads for dynamic approaches. As reported in [42], CUJO takes averagely 500 ms to analyze a webpage in dynamic feature extraction. In JS*, it averagely takes 1 second to generate or replay one trace, for a given script snippet.

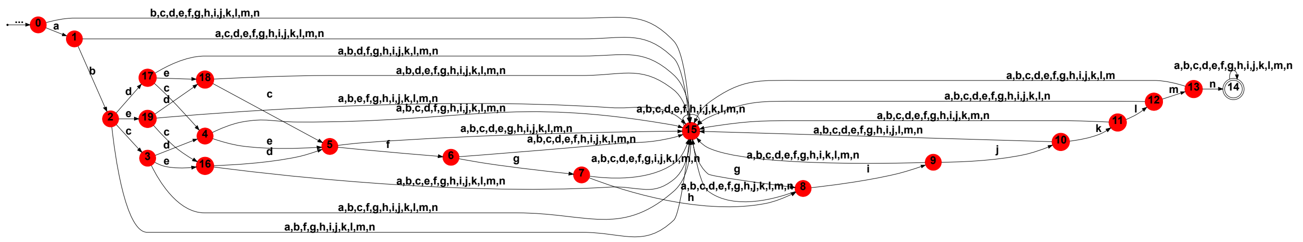
Owning to the step of preprocessing, a small alphabet can be built from the execution traces by filtering security-irrelevant system calls out and extracting common actions from traces. Alphabetical sizes ($|\Sigma|$) of the 8 learned DFAs are all less than 30, and 5 out of 8 DFAs even have $|\Sigma|$ less than 16. Usually, a small value of $|\Sigma|$ leads to a small number of raised membership queries and candidate queries, e.g., DFAs with $|\Sigma| \leq 11$ have **#M-Q**$\leq 42K$ and **#C-Q**$\leq$ 12. For these DFAs with $|\Sigma| \leq 11$, the core learning process can be accomplished in 5 seconds. For other types except *Type VII*, it takes only less than 9 seconds. The most time-consuming one is for *Type VII*. Considering large values of **#M-Q** and **#C-Q**, it is fast to finish core learning in 22 seconds. It is also acceptable to finish all, including trace generation and replay, in 4597 seconds.

Another observation is that JS* requires a large value of **#M-Q** but a small value of **#C-Q**. The explanation is that action sequences in the training set are quite different from each other in terms of length or substring. In contrast, sequences π_{m1} to π_{m6} and π_{b1} to π_{b6} in Fig. 4 show high similarity in length or substring, with different positions of action c. These similar sequences in our running example quickly lead to a closed and consistent observation table, which makes **#M-Q**$=$ 622 and **#C-Q**$=$ 2. However, for these 8 real attack types, there are no such ideally similar sequences that lead to quick convergence to a closed and consistent observation table. Thus, it usually needs a large number of membership queries to reach a candidate DFA.

RQ3: Tool comparison. We compare JS* with the open JS malware detection service JSAND 2.3.6 [7][26] and VIRUSTOTAL [11], an online malware detection service powered by 56 mainstream anti-virus products. The comparison mainly focuses on FN rate and average predication time. We do not compare FP rate, as 1000 benign samples are verified by the union of results from JSAND and 56 tools on VIRUSTOTAL—no FP case in benign samples for JSAND and 56 tools on VIRUSTOTAL.

Totally, JSAND[7] correctly detects 39.1% (61/156) malicious samples, and 95 FN cases are not evenly distributed among the 8 types (see Table 2). Among the 8 attack types, JSAND yields the lowest FN rate (8.7%) for *Type VI*, and the highest FN rate (80%) for *Type I* and *Type II* attack. As JSAND is a dynamic detection tool, due to the limit of the used honeypot, it may miss samples from *Type I*

[7]As JSAND uses on dynamic analysis, we submitted each sample ten times. If any submission reports that the sample is malicious or suspicious, we consider it malicious.



0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
a
b
c
d
e
f
g
h
i
j
k
l
m
n
b,c,d,e,f,g,h,i,j,k,l,m,n
a,c,d,e,f,g,h,i,j,k,l,m,n
a,b,d,f,g,h,i,j,k,l,m,n
a,b,d,e,f,g,h,i,j,k,l,m,n
a,b,e,f,g,h,i,j,k,l,m,n
a,b,c,d,f,g,h,i,j,k,l,m,n
a,b,c,d,e,g,h,i,j,k,l,m,n
a,b,c,d,e,f,h,i,j,k,l,m,n
a,b,c,d,e,f,g,i,j,k,l,m,n
a,b,c,e,f,g,h,i,j,k,l,m,n
a,b,c,f,g,h,i,j,k,l,m,n
a,b,f,g,h,i,j,k,l,m,n
a,b,c,d,e,f,g,h,i,j,k,l,m,n
a,b,c,d,e,f,g,h,j,k,l,m,n
a,b,c,d,e,f,g,h,i,k,l,m,n
a,b,c,d,e,f,g,h,i,j,l,m,n
a,b,c,d,e,f,g,h,i,j,k,m,n
a,b,c,d,e,f,g,h,i,j,k,l,n
a,b,c,d,e,f,g,h,i,j,k,l,m

a	$\Rightarrow$ nsIDOMCrypto.generateCRMFRequest()	f	$\Rightarrow$ nsISocketTransportService.createTransport()	k	$\Rightarrow$ nsIDocShell.busyFlags()
b	$\Rightarrow$ nsIHttpProtocolHandler.userAgent()	g	$\Rightarrow$ nsISocketTransport.openOutputStream()	l	$\Rightarrow$ nsIDocShell.currentDocumentChannel()
c	$\Rightarrow$ nsIDOMNavigator.userAgent()	h	$\Rightarrow$ nsISocketTransport.openInputStream()	m	$\Rightarrow$ nsIHttpChannel.responseStatus()
d	$\Rightarrow$ nsIDOMNavigator.platform()	i	$\Rightarrow$ nsIInputStreamPump.init()	n	$\Rightarrow$ nsIHttpChannel.isNoStoreResponse()
e	$\Rightarrow$ nsIDOMNavigator.buildID()	j	$\Rightarrow$ nsIInputStreamPump.asyncRead()		

Figure 8: The partial DFA of *Type I* attack, which models the exploit to CVE-2013-1710

Table 3: Detection Ratio of our approach and other tools on 156 malicious samples in the testing data set

Tool	Detection %	Tool	Detection %
JS*	95.51%	MCAFEE	57.05%
AVAST!	81.41%	JSAND	39.10%
GDATA	73.72%	TREND	30.77%
AVG	73.08%	SYMANTEC	28.21%
BITDEFENDER	71.15%	CLAMAV	12.82%
F-SECURE	69.87%	PANDA	1.92%
KASPERSKY	67.95%		

and *Type II*, which are platform specific and not easy to trigger.

Column **V.T.**[8] in Table 2 denotes the average number of tools that detect each malicious sample, among 56 tools provided by VIRUSTOTAL. For each of 156 malicious samples, on average 21 (37.5%) of 56 tools can successfully detect it. We also observe that on average 34 (60.7%) of 56 tools can detect each of *Type VI* samples. This observation indicates that 56 tools on VIRUSTOTAL can generally better detect *Type VI* attacks than others. This observation is consistent with previous finding that JSAND has the lowest FN rate (8.7%) for *Type VI*. Among the 8 types, on average only 15 tools (26.8%) can detect each of *Type VII* attacks. Thus, *Type VII* attack is difficult to detect for state-of-the-art tools, due to its flexible attack behaviors and enormous variants. This point is supported by the complexity of the inferred DFA of *Type VII*.

To see the capabilities of state-of-the-art tools, in Table 3, we test the detection ratio of the open-source anti-virus tool CLAMAV [14] and 2014 best reviewed anti-virus products [15]: AVG, AVAST!, BITDEFENDER, F-SECURE, GDATA, KASPERSKY, MCAFEE, PANDA, SYMANTEC and TRENDMICRO. On the testing set of 156 malicious samples, the best tool AVAST! achieves a detection ratio of 81.41%. Other tools perform even worse. We manually inspect FN cases for JS* and other tools. One sample that belongs to *Type VII* is missed by both JS* and VIRUSTOTAL, as it targets at mobile platform and fails to launch the attack in our testing environment.

According to the JSAND's report, we calculate the predication time by deducting the analysis starting time from the report generation time. Averagely, it takes 4.7 seconds for JSAND to finish the execution and predication of one sample (see column **JSAND T(s)** in Table 2). In contrast, JS* takes averagely 1.36 seconds to execute the tested sample and check the trace with 8 learned DFAs. Thus, the predication time was reduced by 71% in JS*. For predication time of the 56 tools, VIRUSTOTAL runs them in parallel and sets a timer (1 minute) to prevent no response. As results from different tools are dynamically added to the result page, according to our observations, most tools can finish the predication in 5-10 seconds.

RQ4: Detecting variants. The predication results show the capability of JS* in detecting malicious variants of the same attack type, as the 120 training samples used for learning share quite low textual similarity with the 156 testing samples. We use code clone detection tool CloneMiner [18] and fail to detect sample pairs that have file-level textual similarity above 30% (due to different exploits and obfuscation). Thus, owing to the nature of dynamic analysis of JS*, syntactic obfuscation poses no challenge to JS*.

Detecting new attack. Besides the DFAs, we also test the usefulness of the partial DFAs. Among the 156 testing samples, an new attack that exploits CVE-2014-1580 is missed by JSAND and other 56 tools. Our 8 inferred DFAs also fail to accept its traces. However, the partial DFA in Fig. 7 can detect it, as the payload execution part of its traces is accepted by the DFA. Thus, the partial DFAs can also be used for fine-grained behavior identification. In this case, payload execution helps to detect the new attack, but cannot classify according to exploit type. Last, based on the partial DFAs, we manually craft 8 malicious samples for each attack types, by combining different exploits and payloads. These new crafted ones can fail existing tools, e.g., on averagely 23 tools on VIRUSTOTAL can detect the crafted, while originally 33 tools detect them averagely. The partial DFAs and crafted samples are also available in [9].

6.4 Discussion

DFA or stateful typestate? To sum up, the experiments have shown the effectiveness of JS* in modelling attack behaviors and predicating variants. JS* generally works well for malicious attacks with clear and representative common attack actions, e.g., *Attack targeting JRE* and *Attack based on multimedia (e.g., images, videos)*. According to our controlled experiments, the traces generated by a handful of unique and representative samples can help JS* to efficiently infer a DFA. Currently, we only uses the arguments of system calls for data analysis, not for argument based behavior modelling. The rationale is twofold: (1) We model JS attack based on system calls, not JS APIs. System calls work at the lower level than JS APIs. The same code snippet (with the same APIs) leads to different low level system call traces. Thus, system call traces essentially are reflection of APIs together their arguments. (2) In reality, these 276 malicious samples mostly can launch the attack with the fixed built-in arguments, without the interaction or input from users. In our previous pilot study, we found argument-based behavior modelling requires *stateful typestate* (DFA with guard conditions for transitions). According to [50], a stateful DFA is good at modelling behaviors of simple data-rich program with only 2 or 3 actions. However, in our study, considering the alphabetical size, inferring typestate is not scalable but prone to path exploration.

DFA decomposition. Currently, we model major behaviors of an attack type in one DFA, including vulnerability exploiting and payloads execution. For each step, there might be several different behavior patterns, e.g., different payloads executions in Fig. 1 (exe-

[8]We also submitted each samples to VIRUSTOTAL five times in Dec. 2014, and the results reported by VIRUSTOTAL were consistent for different submissions.

cuting arbitrary code in victim computer) and Fig. 7 (binding shell using TCP). We initially manually identify such patterns from one DFA. Then we convert patterns and other DFAs into directed cycle graphs, and then we apply GENERICDIFF [51] to do graph matching — checking if these patterns exist in other DFAs. We also extent GENERICDIFF for sub-graph isomorphism to extract possible patterns (similar partial DFAs) from 8 inferred DFAs. Some found patterns are also reported in [9]. In this study, we treat attack behavior modelling in a top-down way — tracking the whole trace of an attack to model and classify. In other study on mining API usage patterns [40], behavior modelling is done in a bottom up way.

6.5 Threats to Validity

There are several threats to validity. First, the inferred models are based on dozens of representative variant traces confirmed by VXHEAVEN, OPENMALWARE and WEB INSPECTOR. Thus, the sample of the collected traces directly affect the learning results. To address this problem, we need to further investigate the impact of sample size and representativeness on the results. Second, the parameters used for the Levenshtein distance similarity in Section 4 are commonly used in code clone similarity analysis [37], i.e., a threshold of 80% similarity in our study. Parameters for *randomWalks* in Algorithm 4 are the same as those used in [50]. Further investigation is needed to see the effects of parameters. Lastly, the external validity is that the results are observed from the 8 specific JS attack types. Due to the greatly distinct natures of different JS attacks, the results of this study may not be applicable to malware whose malicious behavior are not reflected by system call invocation. We remark that this is one assumption of this work.

7. RELATED WORK

System-call based behaviour modelling and malware detection. Existing studies [36][43][49] have utilized system calls in dynamic analysis to detect abnormal behaviours, such as attacks or intrusions. But these studies failed to model program behaviours, until Sekar *et al.* [44] modelled normal executions (system call sequences) of a process as a Finite State Automata (FSA) when most normal traces are available. [44] uses a heuristic algorithm to model normal behaviours rather than learning using L^* algorithm, and the modelling process is not in polynomial time. Kolbitsch *et al.* [35] modelled program behaviour using a direct acyclic graph of system calls along with their data-dependent parameters, and applied graph matching algorithm on these graphs to identify abnormal ones. Christodorescu *et al.* [25] further defined a new graph representation of program behaviour and applied a machine learing algorithm to mine malware specifications from dependence graphs of the malicious and benign programs. Besides, layered system call graph in [39] and tainted argument analysis for selected calls in [46] are proposed for malware detection.

A recent quantitative study [21] investigated how the choice of system-call based behaviour modellig influences the quality of detection results. They found that N-grams is the best model for low-level system calls, whereas bags and tuples without order information yield best results when high-level actions are introduced. However, in [21] other models (e.g., graph or DFA) are not discussed and the authors acknowledged the impossibility of generalizing observed results in a closed form. JS^* uses action sequences to model attack behaviours in the form of DFA. Although orders of actions matter in the DFA, the DFA learned by JS^* accepts the equivalent action sequences derived from the corresponding tuples or bags of actions, owning to the analysis on EPs of action sequences.

Inferring behavior model by L^*. Several studies adopt the L^* to infer behavior, but not attack behaviors. Chia *et al.* [24] proposed to infer botnet command and control protocols from the sequence of messages sent over the network by using the L^* algorithm. Chia *et al.* [23] also presented an approach to infer an abstract model of the analysed application in form of DFA, and to then apply symbolic execution for bounded state-space exploration by virtue of the guidance provided by the inferred DFA. Xiao *et al.* [50] applied L^* on method call sequences in randomly generated testing code to model behaviors as stateful typestate, which is suitable for data-rich programs, i.e., stack, piped stream, etc.

Answering membership queries depends on domain knowledge, e.g., in [50] for behavior learning, if an exception is thrown for the generated testing code, the corresponding method sequence is not accepted; in [23] for protocol inference, if a message exchange trace violates the protocol, the given trace is not accepted. In our study, if malicious results are produced or defense rules are violated, the tested action sequence is accepted. Candidate queries can be answered in three ways. First, the expected DFA to be learned is available, e.g., the existing protocol in [23]. Second, random sampling is used to generate traces to test the equivalence between a candidate DFA and the expected DFA [50]. Last, model checker is applied to verify the equivalence between a candidate DFA with the expected one based on some properties [16].

JavaScript malware detection. The existing studies on detection of malicious JS mainly adopt static analysis, or dynamic analysis, or hybrid analysis to identify the characteristics of malicious JS. JSAND [26] extracted features from 4 aspects (redirection, deobfuscation, environmental context and exploitation) via dynamic analysis, and used Näive Bayes to detect JS malware. Canali *et al.* [20] proposed to perform a large-scale static analysis to identify the malicious web pages by applying a fast and reliable filter PROPHILER. Curtsinger *et al.* [27] presented ZOZZLE, a tool that predicates the benignity or maliciousness of JS code by using features associated with abstract syntax tree (AST) hierarchy information. REVOLVER [34] also heavily relies on static analysis to build the ASTs and to compute the similarity among ASTs. CUJO [42] uses hybrid analysis to on-the-fly extract dynamic and static features from program information and execution traces of JS programs, respectively. All extracted features are processed by *q-grams* for SVM based classification. These studies focus on detection of general JS malware. Other existing studies rely on dynamic analysis to detect specific attacks [22][29][38][41][47]. Compared with these tools, JS^* can model attack behaviours. It does not require a large-size training set and can be generally applicable to attacks with explicit browser-level system calls (actions).

8. CONCLUSIONS

In this work, we propose the approach JS^* for detecting malicious JS via attack behavior modelling. JS^* is based on L^* algorithm and the learned model is in the form of a DFA. Our key contribution is to combine *data dependency analysis*, *defense rules* and *replay mechanism* to implement an online teacher for detection and classification of malicious JS. We evaluate JS^* using eight popular types of attacks. The experimental results demonstrate scalability and effectiveness of our approach as well as the usefulness of the inferred DFAs (or partial DFAs). For the future works, we are planning to extend our approach to general attacks in OS level.

9. ACKNOWLEDGEMENTS

This research is supported by the National Research Foundation, Prime Minister's Office, Singapore, under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate.

10. REFERENCES

[1] Alexa Top Sites. http://www.alexa.com/topsites.

[2] OpenMalware. http://http://oc.gtisc.gatech.edu:8080/.

[3] VXHeaven. http://vxheavens.com/.

[4] Marketscope for AJAX technology and RIA platforms. https://www.gartner.com/doc/847312/, 2008.

[5] The vulnerability of CoolPreviews. http://www.security-assessment.com/files/advisories/CoolPreviews_Firefox_Extension_Security_Advisory.pdf, 2008.

[6] Mozilla Configurable Security Policies. http://www-archive.mozilla.org/projects/security/components/ConfigPolicy.html, 2009.

[7] JSand on-line service. https://wepawet.iseclab.org/index.php, 2012.

[8] Kaspersky security bulletin 2013. http://media.kaspersky.com/pdf/KSB_2013_EN.pdf, 2013.

[9] JS*: Malicious JavaScript Detection via Attack Behavior Modelling. http://pat.sce.ntu.edu.sg/jsstar, 2014.

[10] MDN: Mozilla technologies: XPCOM. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM, 2014.

[11] VirusTotal: an on-line malware detection service . https://www.virustotal.com/, 2014.

[12] Web Inspector. http://www.webinspector.com/, 2014.

[13] Microsoft Security Intelligence Report,Volume 15. http://www.microsoft.com/security/sir/archive/default.aspx, Jan. 2013 to Jun. 2013.

[14] Clamav. http://www.clamav.net/, Jan, 2015.

[15] 2014 best antivirus software review. http://anti-virus-software-review.toptenreviews.com/, Sep, 2014.

[16] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109, 2005.

[17] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[18] H. A. Basit and S. Jarzabek. A data mining approach for detecting higher-level clones in software. *IEEE Trans. Software Eng.*, 35(4):497–514, 2009.

[19] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.

[20] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *WWW*, pages 197–206, 2011.

[21] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. A quantitative study of accuracy in system call-based malware detection. In *ISSTA*, pages 122–132, 2012.

[22] Y. Cao, V. Yegneswaran, P. A. Porras, and Y. Chen. Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In *NDSS*, 2012.

[23] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security Symposium*, 2011.

[24] C. Y. Cho, D. Babic, E. C. R. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. In *ACM Conference on Computer and Communications Security*, pages 426–439, 2010.

[25] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *ESEC/SIGSOFT FSE*, pages 5–14, 2007.

[26] M. Cova, C. Krügel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *WWW*, pages 281–290, 2010.

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

[28] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6, 2012.

[29] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *DIMVA*, pages 88–106, 2009.

[30] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

[31] O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. In *ICECCS*, pages 85–94, 2005.

[32] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[33] M. Johns and J. Winter. Protecting the intranet against javascript malware and related attacks. In *DIMVA*, pages 40–59, 2007.

[34] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An automated approach to the detection of evasiveweb-based malware. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 637–652, Berkeley, CA, USA, 2013. USENIX Association.

[35] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-Y. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX Security Symposium*, pages 351–366, 2009.

[36] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. *IEEE Software*, 14(5):35–42, 1997.

[37] T. Lavoie and E. Merlo. An accurate estimation of the levenshtein distance using metric trees and manhattan distance. In *IWSC*, pages 1–7, 2012.

[38] V. B. Livshits and W. Cui. Spectator: Detection and containment of javascript worms. In *USENIX Annual Technical Conference*, pages 335–348, 2008.

[39] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *RAID*, pages 78–97, 2008.

[40] H. V. Nguyen, H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Mining interprocedural, data-oriented usage patterns in javascript web applications. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 791–802, 2014.

[41] 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.

[42] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient

detection and prevention of drive-by-download attacks. In *ACSAC*, pages 31–39, 2010.

[43] I. Sato, Y. Okazaki, and S. Goto. An improved intrusion detecting method based on process profiling. *IPSJ Journal*, 43(11):3316–3326, 2002.

[44] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, pages 144–155, 2001.

[45] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *WWW*, pages 921–930, 2010.

[46] E. Stinson and J. C. Mitchell. Characterizing bots' remote control behavior. In *DIMVA*, pages 89–108, 2007.

[47] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *EUROSEC*, page 4, 2011.

[48] J. Wang, Y. Xue, Y. Liu, and T. H. Tan. JSDC: A hybrid approach for javascript malware detection and classification. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 109–120, 2015.

[49] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *DSN*, pages 368–377, 2005.

[50] H. Xiao, J. Sun, Y. Liu, S.-W. Lin, and C. Sun. Tzuyu: Learning stateful typestates. In *ASE*, pages 432–442, 2013.

[51] Z. Xing. Model comparison with genericdiff. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 135–138, 2010.

[52] W. Xu, F. Zhang, and S. Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 9–16. IEEE, 2012.

[53] P. Zeng, J. Sun, and H. Chen. Insecure javascript detection and analysis with browser-enforced embedded rules. In *PDCAT*, pages 393–398, 2010.