Detection and Classification of Malicious JavaScript via Attack Behavior Modelling

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

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 approaches. 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 commonly adopted by 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 JSEND [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. Suspicous 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 modelling 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 whether 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...
The prevalence of JS has attracted attackers to employ it for their malicious intentions. Malicious JS takes advantages of vulnerabilities (e.g., Trojan.js.frame.* 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 JSs, we perform a preprocessing to simplify these traces by removing security-irrelevant system calls. The simplified (abstracted) traces are called action sequences. Second, we develop an online learning algorithm based on the 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 show that JS∗ does not only outperform existing commercial solutions in malware detection, but also detects 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 malicious intentions. Malicious JS takes advantages of 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

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

```
<%a href='javascript: var file=Components.classes["@mozilla.org/file/local;1"];
createInstance(Components.interfaces.nsILocalFile);
file= newURI("data:text/html;base64,PHN2ZWN0IG5vdGVzLjAp...; | void
⇒ nsIIOService2 | newURI | 3 | data:text/html;base64,PHN...; | void
⇒ nsIURI | scheme | 0 | | void
⇒ nsIPrefBranch | getCharPref | 1 | preview.enable; | void
⇒ nsIPrefBranch | getBoolPref | 0 | devtools.inspector.enabled; | void
⇒ nsIPrefBranch | getCharPref | 1 | preview.enable; | void
⇒ nsIOService | newURI | 1 | data:text/html;base64,PHN...; | void
⇒ nsIProcess | run | 2 | [path],1; | void
⇒ nsIProcess | init | 1 | object; | void
⇒ nsIProcess | init | 1 | object; | void
⇒ nsIIOService | newURI | 3 | data:text/html;base64,PHN...; | void
⇒ nsIPrefBranch | getCharPref | 1 | preview.enable; | void
⇒ nsIPrefBranch | getBoolPref | 0 | devtools.inspector.enabled; | void
⇒ nsIPrefBranch | getCharPref | 1 | preview.enable; | void
⇒ nsIOC | init | 1 | object; | void
⇒ nsIOC | init | 1 | object; | void
⇒ nsIOC | init | 1 | object; | void
⇒ nsIOC | init | 1 | object; | void
⇒ nsIOC | init | 1 | object; | void
⇒ nsIOC | init | 1 | object; | void
```

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 \( s = (I, M, N_p, S_p, T_r) \), where \( I \) is the interface name of \( s_c, M \) is the method name, \( N_p \) is the list 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 \( \pi = (s_{c1}, s_{c2}, \ldots, s_{cn}) \), 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 zone1. 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 ‘\( \Rightarrow \)’. The trace shown on the right column only contains I2 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

---

1The 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 process 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 by constructing a total blank web page. We consider system calls to these 667 security-irrelevant system mon interfaces as no security risks, i.e., typically used by system level applications to construct a new action sc, then

Algorithm 1: extractCommonActions

input : $S_e = \{\pi_1, \pi_2, ..., \pi_n\}$, 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 $\pi \in S_e$ do
2. foreach system call $sc \in \pi$ do
3. if $\exists a_k \in M.$keys • 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. $\Sigma \leftarrow \Sigma \cap GetAction(\pi_1, M)$
9. foreach $\pi \in \{\pi_2, ..., \pi_n\}$ 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 attacks, 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 XPOM 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 XPOM [10]) basic and very common interfaces as no security risks, i.e., security-irrelevant system calls, e.g., system call nsIDOMWindow.getScrollRect 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 nsIPrefBranch.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; nsIService1.newURI and nsIService2.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 multithreading 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 $S_e$. First, the map $M$ is built from lines 1 to 7. At line 3, each system call $sc$ in $\pi$ from $S_e$ is compared with each key $a_k$ in $M$ by calling IsSameAction to check the functionality...
that an attack model should use two types of queries: membership queries and candidate queries. The former asks whether a given trace is accepted by the black box DFA \( \mathcal{D} \), 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 \( \mathcal{D} \).

### 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; \( \Sigma \) is the finite set of input events—so called alphabets; \( \delta \) 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 \( \Sigma \) 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 \( \mathcal{D} \).

### 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 \( \pi_a \) 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: isEqualEqualPermutation

```
input : S_e = \{π_1, π_2, ..., π_n\}, a set of malicious action sequences
input : π, a given action sequence
output: true or false, if π is an equivalent permutation
1 S_e ← ∅;
2 foreach sequence πᵢ ∈ S_e do
3   S_eₜ ← getDependencyClosure(πᵢ);
4   Sₑₑ ← Sₑₑ ∪ getPermutations(πᵢ, Sₑₜ);
5 if π is in Sₑₑ 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][33]). 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 πᵢ = (a₁, a₂, a₃), where a₂ has data dependency on a₁ and a₂, denoted as (a₃ → (a₁, a₂)), we can infer π₂ = (a₂, a₁, a₃) is also malicious. So π₂ and πᵢ are EPs.

Given a training set of malicious sequences Sₑₑ and an unknown system call sequence π, Algorithm 3 shows how to test if π is an EP according to the sequences in Sₑₑ. The basic idea is to get the data dependency closures among actions inside a sequence πᵢ by invoking method getDependencyClosure(πᵢ). In two steps, this method builds the dependency closure³. Given πᵢ, 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 ← a\} and \{(c, f) ← d\}. As c is independent, actually, πᵢ₃ to πᵢ₆ (and other similar sequences only with different c positions) are all EPs.

By getPermutations, the mutually independent relation between actions (e.g., a₁ and a₂ in the above case of π₁, c and other actions)

³The data dependency is calculated based on the original system calls of the actions, we omit the implementation details for the interest of space.

⁴A dependency closure is a transitive relationship among actions, and each in this closure is directly or indirectly involved in data dependencies on others.

![Figure 5: The JEIS statements reverse-engineered from action sequence π₉₁ \{ (c.a.b.d.e.f) \}](image)

Figure 5: The JEIS statements reverse-engineered from action sequence π₉₁ \{ (c.a.b.d.e.f) \} in the case of π₉₁ infers the equivalent permutations. All EPs should not equal to an existing benign sequence, and then be stored into a set for comparison with πᵢ. A right match indicates an EP. Practically, the calculation of Sₑₑ 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 ← m ← l ← k ← j ← i ← (g, h) ← f ← (c, d, e) ← b ← 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).

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 \{a.b.d.e.f\} (removing prefix c from π₉₁) and \{c.a.b.d.e.f\} (partial reordering of π₉₁), 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 inferred 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 \{a.b.d.e.f\} (removing prefix c from π₉₁). To replay \{a.b.d.e.f\}, we get the reverse-engineered JEIS of π₉₁, then remove the intermediary script relevant to c, and finally craft the expected JEIS that contains statements or crashes. Such exceptions or crashes indicate that the replayed sequence is infeasible rather than malicious, e.g., according
to JEIS of $\pi_{m1}(c.a.b.d.e.f)$ in Fig. 5, we want to craft the JEIS of $\{c.a.b.d.e.f\}$ 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 nslProcess.run (statement 9) is executed before nslProcess.init (statement 8). So $\{c.a.b.d.e.f\}$ 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 $S_o$ and benign sequences $S'_o$, e.g., the sequences in Fig. 4, Algorithm 2 describes the process in answering the membership query regarding to the given action sequence $\pi_o$.

First, if any $\pi_i \in S_o$ is a prefix of $\pi_o$ (line 1), i.e., $\pi_o$ equals to or starts with $\pi_i$, $\pi_o$ must be malicious. If $\pi_o$ equals any sequence in $S'_o$ (line 3), $\pi_o$ must be benign. Then isEqualPermutation($\pi_o,S_o$) method at line 5 defined in Algorithm 3 is called to check whether $\pi_o$ is an EP, and a true answer means that $\pi_o$ is also a malicious EP.

The next step is to check if $\pi_o$ violates the defense rules in Section 5.1.1 at line 7. Finally, method Replay($\pi_o$) at line 9 is called to concretely execute $\pi_o$ 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 since it starts with an existing malicious sequence $\pi_{m1}$. An example for isEqualPermutation($\pi_o,S_o$) at line 5 is to check action sequences $\langle c.a.b.e.f.d \rangle$ and $\langle a.b.c.e.d.e.f \rangle$. According to the collected traces like $\pi_{m2}$ 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 $\langle c.a.d.b.e.f \rangle$. But $\langle a.b.c.e.d.e.f \rangle$ is malicious as it satisfies the identified data dependency, similar to $\pi_{m3}$ 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($\pi_o$) at line 9 is effective in running the left uncertain sequences ranging from simple ones ($a.b$) to those complicated ones like $\langle b.a.f.a.b.f.c.e.d.a.e.d.d.f \rangle$ for membership querying. Actually, Replay($\pi_o$) 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 $C$ is inferred after multiple membership queries. To judge if $C$ is equivalent to the expected DFA $D$ 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 $\pi$—false positives and false negatives. The former means that a sequence accepted by $C$ should be rejected by $D$, while the latter means that a sequence rejected by $C$ should be accepted by $D$.

Given $S_o$ and $S'_o$ in Fig. 4, Algorithm 4 illustrates how the teacher answers the candidate query for the given $C$, based on regression testing and random sampling testing. First, at line 1, each sequence $\pi_i$ from the known sequence set $S_o$, $S'_o$ and the previously tested sequence set $S_{m2}$ is input to membershipQuery and also $C$, isAccepted(). Method membershipQuery($\pi_i$) at line 2 returns true if $\pi_i$ is accepted by $D$. If $C$ and $D$ show different acceptance results for $\pi_i$, a counterexample $\pi_i$ is found and returned. Second, a random walk function randomWalks() is used to generate $S_{n2}$, a new set of random sequences that include both accepted and rejected ones on $C$. The rationale of using random walk is that sequences on $C$ cannot be enumerated due to potential loops. randomWalks() at line 4 has three parameters, where $C$ is the candidate DFA; $C$.stateSize*times denotes the number of generated sequences and times is an input constant to multiply; $C$.stateSize+extraLenLmt is the maximum allowed length of generated sequences and extraLenLmt is the extra length that can be larger than $C$.stateSize. Here, $C$.stateSize denotes the size of total states in $C$. Then, each sequence $\pi_i$ in $S_{n2}$ is also given to membershipQuery() and $C$.isAccepted() for acceptance check—an inconsistency indicates a counterexample. Note that any tested $\pi_i$ from $S_{n2}$ is added to $S_{m2}$, which is used for regression testing in answering the next candidate query. Finally, if no counterexample is found, $\pi_{ce}$ is returned and the learning process stops.

In our running example, candidate queries are asked twice. The first candidate DFA $C_1$ in Fig. 6 (a) is inferred when $P$ in the observation table only contains $\{a,b,c,d,e,f\}$ with one state 0. For $C_1$, 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 $C_2$ in Fig. 6 (b) is learned. According to regression testing and random sampling testing in Algorithm 4, no counterexample for $C_2$ is found—$C_2$ is equivalent to the expected $D$.

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 $D$ to model the attack—or equivalently a regular language $\mathcal{L} = (e)^* \cdot a \cdot (e)^* \cdot b \cdot (e)^* \cdot d \cdot (d)^* \cdot e \cdot (e)^* \cdot f \cdot (c)^*$ 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 $\pi$ over $\Sigma$. An acceptance of $\pi$ on $D$ suggests that $\pi$ is from a malicious trace. The sequences in $S_o$ 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 $\pi_{m1}$ the last argument of action $f$ is “0”, which makes nslProcess.run include zero argument from the argument list. One way to solve the problem is

<table>
<thead>
<tr>
<th>Algorithm 4: candidateQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong>: $C \neq \text{NULL}$, the learned candidate DFA</td>
</tr>
<tr>
<td><strong>input</strong>: $\pi_o$, the set of old sequences that have been tested in previous calls of candidateQuery, initially being $\emptyset$ before any candidateQuery is called</td>
</tr>
<tr>
<td><strong>output</strong>: $\pi_{ce}$, an counterexample sequence found</td>
</tr>
<tr>
<td>1. foreach $\text{trace } \pi_i \in (S_o \cup S'<em>o \cup S</em>{m2})$ do</td>
</tr>
<tr>
<td>2. if $\text{membershipQuery}(\pi_i) \neq C$.isAccepted($\pi_i$) then</td>
</tr>
<tr>
<td>3. return $\pi_{ce} \leftarrow \pi_i$;</td>
</tr>
<tr>
<td>4. $S_{n2} \leftarrow \text{randomWalks}(C,C$.stateSize$ \times \text{times},C$.stateSize$+\text{extraLenLmt})$;</td>
</tr>
<tr>
<td>5. foreach $\text{trace } \pi_i \in S_{n2}$ do</td>
</tr>
<tr>
<td>6. $S_{m2} \leftarrow S_{m2} \cup {\pi_i}$;</td>
</tr>
<tr>
<td>7. if $\text{membershipQuery}(\pi_i) \neq C$.isAccepted($\pi_i$) then</td>
</tr>
<tr>
<td>8. return $\pi_{ce} \leftarrow \pi_i$;</td>
</tr>
<tr>
<td>9. return $\pi_{ce} \leftarrow \text{NULL};$</td>
</tr>
</tbody>
</table>
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 $\pi_{n1} : \langle c, a, b, d, e, f(0) \rangle$. To refine the DFA for such case, we derive new actions $f_m$ and $f_n$ from current action $f$. Here, $f_m$ refers to the action $f(\{s_1, \ldots, s_n\})$ with the argument set $\{s_1, \ldots, s_n\}$ that produces malicious outcome like $f(1)$. Similarly, $f_n$ refers to the action $f$ with arguments that produce benign outcome. Then with the new $\Sigma' = \{a, b, c, d, e, f_m, f_n\}$, 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 attacks 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 lower 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 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 samples from the Alexa [1] top 100 web sites, none of which is reported as malicious by the 56 tools provided by VIRUS TOTAL.

Among the total 276 malicious samples, we select 120 samples (≈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 execution. 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 $\Sigma$ briefly tells whether common essential behaviors of the same attack type are captured and modelled. From our observation, the set of common actions $|\Sigma|$ 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 nsFile.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 observe that the eight inferred DFAs share some common parts, e.g., payload executions. 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

| Attack Type | $|S|$ | $|\Sigma|$ | $#M-Q$ | $#C-Q$ | Time(s) | Total Time(s) |
|-------------|-----|-------|--------|--------|---------|--------------|
| Type I      | 67  | 29    | 4      | 1.83   | 945     |              |
| Type II     | 77  | 11    | 33,585 | 0.501  | 364     |              |
| Type III    | 54  | 11    | 41,459 | 4.137  | 902     |              |
| Type IV     | 74  | 28    | 158,120| 1.95   | 1,068   |              |
| Type V      | 121 | 29    | 468,124| 8.33   | 2,768   |              |
| Type VI     | 50  | 11    | 34,266 | 0.551  | 498     |              |
| Type VII    | 466 | 15    | 390,277| 22.425 | 4,597   |              |
| Type VIII   | 74  | 11    | 28,480 | 0.563  | 993     |              |
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 8 contains only 14 actions and 19 states. Thus, our inferred attack behaviors are drive-by-download attacks that yield the lowest FP rate (8.7%) for Type VII attack. As JS* is a dynamic detection tool, due 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 (|Σ|) of the 8 learned DFAs are all less than 30, and 5 out of 8 DFAs even have |Σ| less than 16. Usually, a small value of |Σ| leads to a small number of raised membership queries and candidate queries, e.g., DFAs with |Σ| ≤ 11 have #M-Q < 42K and #C-Q ≤ 12. For these DFAs with |Σ| ≤ 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_{a_1} to π_{a_n} and π_{b_1} to π_{b_m} 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.

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.

Owing 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 (|Σ|) of the 8 learned DFAs are all less than 30, and 5 out of 8 DFAs even have |Σ| less than 16. Usually, a small value of |Σ| leads to a small number of raised membership queries and candidate queries, e.g., DFAs with |Σ| ≤ 11 have #M-Q < 42K and #C-Q ≤ 12. For these DFAs with |Σ| ≤ 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_{a_1} to π_{a_n} and π_{b_1} to π_{b_m} 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.

Table 2 shows 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.

Owing 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 (|Σ|) of the 8 learned DFAs are all less than 30, and 5 out of 8 DFAs even have |Σ| less than 16. Usually, a small value of |Σ| leads to a small number of raised membership queries and candidate queries, e.g., DFAs with |Σ| ≤ 11 have #M-Q < 42K and #C-Q ≤ 12. For these DFAs with |Σ| ≤ 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_{a_1} to π_{a_n} and π_{b_1} to π_{b_m} 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.

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

<table>
<thead>
<tr>
<th>Type</th>
<th>JS* FN</th>
<th>JS* FP</th>
<th>JSAND FN</th>
<th>JSAND FP</th>
<th>VIRUSTOTAL FN</th>
<th>VIRUSTOTAL FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>43/10000</td>
<td>7/156</td>
<td>43/156</td>
<td>4.8/156</td>
<td>31/1235</td>
<td>3.1/1235</td>
</tr>
<tr>
<td>Type I</td>
<td>9/10000</td>
<td>0/15</td>
<td>12/15</td>
<td>5.9/15</td>
<td>9/15</td>
<td>3.3/15</td>
</tr>
<tr>
<td>Type II</td>
<td>3/10000</td>
<td>0/18</td>
<td>13/18</td>
<td>4.2/18</td>
<td>2/1223</td>
<td>3.4/1223</td>
</tr>
<tr>
<td>Type IV</td>
<td>7/10000</td>
<td>1/15</td>
<td>9/15</td>
<td>3.3/15</td>
<td>11/10000</td>
<td>3.4/11000</td>
</tr>
<tr>
<td>Type V</td>
<td>8/10000</td>
<td>0/23</td>
<td>2/23</td>
<td>8.3/23</td>
<td>8/10000</td>
<td>8.3/10000</td>
</tr>
<tr>
<td>Type VI</td>
<td>3/10000</td>
<td>2/17</td>
<td>18/17</td>
<td>3.4/17</td>
<td>21/7104</td>
<td>3.4/7104</td>
</tr>
<tr>
<td>Type VIII</td>
<td>9/10000</td>
<td>2/27</td>
<td>0/27</td>
<td>5.3/27</td>
<td>9/10000</td>
<td>5.3/10000</td>
</tr>
</tbody>
</table>

55

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.

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.
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

<table>
<thead>
<tr>
<th>Tool</th>
<th>Detection %</th>
<th>Tool</th>
<th>Detection %</th>
</tr>
</thead>
<tbody>
<tr>
<td>JS*</td>
<td>95.51%</td>
<td>McAfee</td>
<td>57.05%</td>
</tr>
<tr>
<td>AVAST !</td>
<td>81.41%</td>
<td>JSAND</td>
<td>39.10%</td>
</tr>
<tr>
<td>GDATA</td>
<td>73.72%</td>
<td>TREND</td>
<td>30.77%</td>
</tr>
<tr>
<td>AVG</td>
<td>73.08%</td>
<td>SYMANTEC</td>
<td>28.21%</td>
</tr>
<tr>
<td>BITDEFENDER</td>
<td>71.15%</td>
<td>CLAMAV</td>
<td>12.82%</td>
</tr>
<tr>
<td>F-SECURE</td>
<td>69.87%</td>
<td>PANDA</td>
<td>1.92%</td>
</tr>
<tr>
<td>KASPERSKY</td>
<td>67.95%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.4 Discussion

**DFA or stateful typestate?** To sum up, the experiments have shown the effectiveness of JS' in modelling attack behaviors and predication 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 use 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 (exc-
cutting arbitrary code in victim computer) and Fig. 7 (binding shell using DFA). 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 exten 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 behaviour 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], behaviour 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. However, 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 behaviors rather than learning using the 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 learning 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 modelling influences the quality of detection results. They found that n-gram 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. J$^*$ uses action sequences to model attack behaviours in the form of DFA. Although orders of actions matter in the DFA, the DFA learned by J$^*$ 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, a 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 (redirect, defacement, environmental context and exploitation) via dynamic analysis, and used Naïve 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. Curtisinger 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. CJO [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, J$^*$ 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 an approach, J$^*$, for detecting malicious JS via attack behavior modelling. J$^*$ is based on the L* algorithm and the learned model is in the form of 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 J$^*$ 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


