JSDC: A Hybrid Approach for JavaScript Malware Detection and Classification

Junjie Wang† Yinxing Xue† Yang Liu† Tian Huat Tan‡
†Nanyang Technological University
‡Singapore University of Technology and Design

ABSTRACT

Malicious JavaScript is one of the biggest threats in cyber security. Existing research and anti-virus products mainly focus on detection of JavaScript malware rather than classification. Usually, the detection will simply report the malware family name without elaborating details about attacks conducted by the malware. Worse yet, the reported family name may differ from one tool to another due to the different naming conventions. In this paper, we propose a hybrid approach to perform JavaScript malware detection and classification in an accurate and efficient way, which could not only explain the attack model but also potentially discover new malware variants and new vulnerabilities. Our approach starts with machine learning techniques to detect JavaScript malware using predicative features of textual information, program structures and risky function calls. For the detected malware, we classify them into eight known attack types according to their attack feature vector or dynamic execution traces by using machine learning and dynamic program analysis respectively. We implement our approach in a tool named JSDC, and conduct large-scale evaluations to show its effectiveness. The controlled experiments (with 942 malware) show that JSDC gives low false positive rate (0.2123%) and low false negative rate (0.8492%), compared with other tools. We further apply JSDC on 1,400,000 real-world JavaScript with over 1,500 malware reported, for which many anti-virus tools failed. Lastly, JSDC can effectively and accurately classify these detected malwares into either attack types.

1. INTRODUCTION

JavaScript is a widely-used client-side scripting language that provides active and dynamic content on the Internet. According to Microsoft's recent security report (Figure 81 in [12]), the prevalence of JavaScript leads to the largest number of malware detected by Microsoft in the first half year of 2013. Most of users typically rely on signature-based anti-virus products to detect malware. However, signature-based detection is not accurate, as it is neither resistant to obfuscation nor applicable to evolving variants of the same malware.

Detection of malicious JavaScript code can be classified into two lines: dynamic approaches and static approaches. Dynamic approaches are mostly based on low-interaction honey clients [25] and high-interaction honey clients [37, 34]. The problem of using honey client lies in the difficulty of setting up a virtual environment that captures the exploitations of all possible plug-ins. Most importantly, honey clients are extremely resource intensive, and it is not affordable to scan millions of scripts. Static approaches mainly adopt machine learning techniques to capture characteristics of malicious scripts. Most of them use obfuscated text information ([16, 27]), syntax information (e.g., AST [18, 26]) or (partial) dynamic information (e.g., API call [17, 33]) as the predicative features for classifying malicious and benign code.

Static and dynamic approaches both have merits and drawbacks. Dynamic approaches are effective, but not scalable. They are usually designed for specific attack types, not for general malware detection. Static approaches based on machine learning techniques or program similarity analysis are efficient, but with high false negative ratio. Besides, machine learning based detection can neither be used to model attack behaviors nor identify new attacks from emerging malware.

To our best knowledge, none of the aforementioned tools go beyond the mere detection of JavaScript malware. Towards an insightful summary and analysis of malware evolution trend, it is desirable to detect malware and further classify them according to the exploited *attack vector* and the corresponding *attack behaviors*. Considering the emergence of numerous new malware or their variants, knowing the attack type and the attack vector will greatly help domain experts for further analysis. Such an automated classification can significantly speed up the overall response to the malware and even shorten the time to discover the zero-day attacks or unknown vulnerabilities.

In this paper, we propose an automatic approach to perform JavaScript malware detection and classification by combining static analysis (via machine learning) and dynamic analysis to achieve both scalability and accuracy. To efficiently detect and classify malware, we propose a two-phase classification. The first phase is to classify malicious JavaScript samples from benign ones, while the second phase is to discover the attack type that a malicious script belongs to. In the first phase, we extract and analyze features that are generally predictive for malicious scripts. To address the obfuscation and dynamic code generation problems, we ex-

ASIA CCS'15, April 14–17, 2015, Singapore..

http://dx.doi.org/10.1145/2714576.2714620.

tend HtmlUnit [1] to obtain the final unpacked code. We extract features from document elements, source code and sensitive function call patterns. In the second phase, we focus on the features that are representative and unique for each attack type. To identify unique features relevant to a certain attack type, the inter-script analysis is adopted to find API usage patterns, which can serve as features of different attack types. For the suspicious candidates that fall into grey zone (uncertain in classification), we adopt the dynamic analysis to unveil the practical behaviors of the attack and match it to existing attack behavior models. For dynamic analysis, we instrument Firefox to capture the attack behaviors of malicious code. Holistically, static analysis assures the scalability while dynamic analysis on uncertain scripts improves the accuracy.

To sum up, this paper makes the following contributions:

- We propose an effective machine learning approach for JavaScript malware detection with predictive features like commenting style based on textual analysis and the feature of function calls with security risks based program analysis.
- We propose a combined approach for JavaScript malware classification by firstly using the machine learning approach with predictive features like function call patterns and the times of external references, and then further improving the accuracy using dynamic program analysis. Our approach achieves the scalability and accuracy.
- We develop the semantic models of the eight known JavaScript attacks in the form of Deterministic Finite Automata, which can not only help to explain the attack behavior, but also discover new malware variants and new vulnerabilities if the attack does not fall into any of the eight attack models.
- We implement out approach in a complete toolset and evaluate it using 20,000 web sites with 1,400,000 scripts. The result shows that our approach beats most mainstream anti-virus tools, with high accuracy and promising performance.

2. JAVASCRIPT MALWARE CATEGORIZATION

Computer Antivirus Research Organization gives each new discovered JavaScript malware a unique family name for identification, e.g., *TrojWare.JS.Agent.G*. Some organizations adopt it while others may have their own naming conventions. Thus, classification information provided by different organizations are neither congruent nor accurate to describe the attacks that the malware can launch. Besides, family-based classification provides no hints on the vulnerability exploited by the malware. A meaningful classification according to the vulnerability and the corresponding exploits is critical, especially for identifying new attacks. Such classification should be based on attack behaviors of the malware that exploit the vulnerability.

In accordance with Kaspersky Security Bulletin Overall Statistics 2013 [8], attacks targeting JRE, Adobe Acrobat Reader, browser, Adobe Flash and so on account up more than 95% of attacks launched by JavaScript code. Attack vector could be a good standard to classification, since it indicates location of vulnerability and the same type shares similar behavior pattern. We list the eight most common JavaScript attack types according to attack vectors as follows.

- *Type I: Attack targeting browser vulnerabilities.* This type of attack targets potential vulnerabilities (like CVE-2014-1567) of a browser of certain version or with certain plug-ins, and then exploits such vulnerabilities to trigger memory corruption. Consequently, it will lead to the malicious control of instruction registers.
- *Type II: Browser hijacking attack.* Browser hijacking is the modification of a web browser's settings performed without the user's permission. A browser hijacker may replace the existing home page, error page, or search page with its own. For instance, CVE-2007-2378 in the Google Web Toolkit (GWT) framework is a JavaScript hijacking attack, which illegally tracks personal data.
- *Type III: Attack targeting Adobe Flash.* ActionScript used in Adobe Flash and Adobe Air has a flaw of allowing chunks allocation in the heap. An exploit page uses vulnerabilities like CVE-2012-1535 to manipulate the heap layout, which induces heap spray attacks. Finally, the memory corruption results in running a ROP (Return Oriented Programming) chain.
- *Type IV: Attack targeting JRE.* Attacks take advantage of vulnerabilities in Oracle's J2SE JDK and JRE. The JVM security depends on the byte code verifier, the class loader, and the security manager. For instance, the vulnerability in Java 7 Update 11 (CVE-2013-1489) allows attackers to bypass all security mechanisms in the Java browser plug-in.
- *Type V: Attack based on multimedia.* Attacks are carried by multimedia files or files in other format supported by browsers, e.g., CSS based attacks. Generally, malicious downloadable resources (e.g., image and font files) can be deliberately designed to exploit vulnerabilities like CVE-2008-5506 to consume the network capacity of visitors, launch CSRF attacks, spy on visitors or run distributed denial-of-service (DDoS).
- *Type VI: Attack targeting Adobe PDF reader.* Vulnerability in Adobe Reader is being used by active attacks targeting individuals with malicious PDF files. For example, vulnerabilities like CVE-2011-2462 could cause a crash and potentially allow an attacker to take control of the affected system.
- *Type VII: Malicious redirecting attack.* Cybercriminals are constantly thinking up new ways to redirect unsuspecting visitors to their drive-by landing page. It includes "Meta Refresh redirecting", "JavaScript Redirect", "CSS redirecting", "OnError redirecting" and so on. This attack type is generally based on CWE-601.
- *Type VIII: Attack based on Web attack toolkits, e.g. Blacole.* Attack toolkits are increasingly available to an unskilled black market that is eager to participate in the speedy spread of malware. According to a recent report [2], Symantec reported that 61% of observed web-based threat activity could be directly attributed to attack kits.

Each attack type aims at representing a unique attack behavior that exploits a certain attack vector or share some certain event routine, e.g., attack type III, IV , V and VI

scan the vulnerabilities in different origins. The above classification is at a more general concept level than family-based classification. For example, the popular malware family *Trojan.JS.Blacole* belongs to the attack type VIII, while the family *Trojan.JS.Iframe* is a malware of type VII. However, compared with the general term of attack type like *heap spray attack* [32], our classification is more specific as we consider attack vector of a heap spray. A heap spray attack can be an attack of type III, IV or V.

3. SYSTEM OVERVIEW OF JSDC

In this section, we sketch the overall workflow of our two-phase approach: JSDC, as depicted in Figure 1. The first phase is to identify the malicious (including highly suspicious) JavaScript snippets. Given malicious JavaScript snippets, the second phase is to further classify them according to their attack type. Thus, the classification process is applied twice, based on the features for malware detection and attack type classification. For the second classification, if there still exist any suspicious fragments that fall into the grey zone (the probability of a script being a certain attack type is non-dominant), dynamic confirmation is applied to execute this script and to capture its execution trace for further analysis.

We use the crawler Heritrix [29], to download web pages for detection. After Heritrix downloads a complete web page with HTML and JavaScript, we extract both internal and external JavaScript snippets. For the ease of feature extraction based on program information, we use HtmlUnit [1], a GUI-Less emulator of a full browser environment, which has fairly good JavaScript support, to obtain the unpacked code (e.g., the code dynamically generated during execution). Once we address the problem of obfuscation and dynamic code generation, we extract features from the source code and AST of the final unpacked code. Note that HtmlUnit is just used to parse and interpret the JavaScript code, without doing the actual rendering job. Thus, this step is regarded as mostly-static analysis [18].

With the training set of representative malicious and benign samples, we propose features that are predictive of malicious or benign intent (see Section 4). Firstly, textual features of the code are extracted based on textual analysis—word size, n-gram model, frequency of characters, commenting style, and entropy are extracted. After textual features are extracted, we use HtmlUnit to obtain the unpacked code, from which features about the program information can be extracted. The program information includes HTML properties and Document Object Model (DOM) operations, and the API usage patterns (i.e., patterns of function calls) in the JavaScript snippet. In addition, we also extract AST-based features, e.g., the tree depth, the pair of type and text from some AST nodes [18]. All these non-textual features are extracted to characterize the malicious behaviors.

To further distinguish eight different types of attacks, we reuse part of features about the program information for detection. Besides, we propose new features based on the vulnerability exploits or attack behaviors of different attack types. We identify the frequently used functions in different attack types. By considering parameters of these functions, we derive the features based on the pair of function and its parameters type. Additionally, features on the API usage patterns and inter-script references are also used.

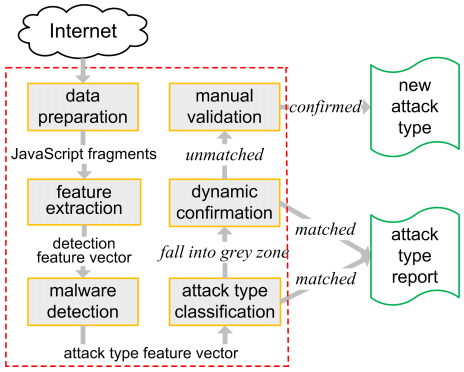

Figure 1: The work-flow of JSDC

Before the actual application of our approach in detection, we need to train the classifiers for predication. We have trained and evaluated each of the used classifiers using k-fold cross validation (CV). The usage of CV is mainly in settings where our goal is prediction, and we aim to estimate the accuracy of a predictive model for malware detection and classification. For the first classifier for malware detection, the training set includes both benign and malicious JavaScript snippets, and each snippet is labelled as malicious or benign. For the second classifier for malware classification according to attack type, the training set only includes malicious snippets and each of them is labelled with a known attack type.

During the process of classifying malicious scripts into different attack types, there are some uncertain results that fall into the grey zone (the classification results show that the probability of a script being a certain attack type is not significantly different from that of being others). This indicates that such scripts are not really similar to any existing malicious samples. For those uncertain ones, a further dynamic analysis is required to execute them and to capture their execution traces at runtime. We have some predefined or learned behaviour models from execution traces of the existing types of attack. More details on inferring attack models in DFAs and checking malware candidate with these models can be found in Section 5.

4. DETAILS OF FEATURE EXTRACTION

JavaScript owns dynamic features that incur security risks and make static analysis fail, e.g., no type checking, DOM based cross-cite scripting, client side open-redirecting and the dynamic code of Ajax [22]. JavaScript's flexibility is reflected in modifying everything ranging from an object's fields and methods to its parent object. Besides, JavaScript's dynamics allow to access and modify shared objects, and to dynamically run the injected code via functions like `eval()`, `setTimeout()`.

Obfuscation is widely adopted to hide the semantics of the JavaScript code from the human efforts' check. Benign JavaScript applications make use of obfuscation to protect intellectual property, while malicious JavaScript uses obfuscation (even multiple levels of obfuscations) to hide its true intent. The flexibility, dynamic features and obfuscation have made the malicious JavaScript code unreadable for human experts. Meanwhile, these characteristics also fail the purely-static approaches to detect malware accurately.

To extract representative features that cover the above characteristics of JavaScript, we adopt three types of analy-

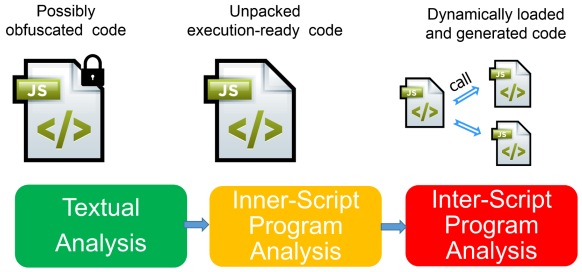

Figure 2: Feature extraction process

sis: textual, inner-script and inter-script analysis (see Figure 2). Textual analysis provides hints of the obfuscated attacks (see Section 4.1). Inner-script analysis (see Section 4.2) mainly provides program information on ASTs or function calls with security risks. Function call patterns and inter-script analysis (see Section 4.3) focus on the classification of attack types. The malicious sample s_1 in Figure 3, which is an obfuscated type I attack that loads external scripts, is used as an illustrative example.

Note that we annotate features only for detection with †, features only for attack type classification with ‡, and features for both purposes with ∗.

4.1 Textual Analysis

Textual analysis is applied to indicate both obfuscation and maliciousness, as it can tell the difference between obfuscated malicious scripts and obfuscated benign ones [16].

Longest word size†. Obfuscated JavaScript code usually contains long words, which signals the existence of encoding or encryption that is used for parameters of the function like eval. A script with very long word size (e.g., larger than 350 words [16]) is likely to be obfuscated. For example, in the obfuscated code in Figure 3(a), the size of the longest word is 814. After tokenization, the longest token size is calculated.

Entropy†. Entropy is a measure of unpredictability of information content, and it is used to analyze the distribution of different characters. Entropy is calculated as follows:

$$H(X) = -\sum_{i=1}^{N}(\frac{x_i}{T})log_{10}(\frac{x_i}{T}) \begin{cases} X = \{x_i, i = 0, 1, ..., N\} \\ T = \sum_{i=1}^{N} x_i \end{cases} \quad (1)$$

where x_i is the count of each characters and T counts all characters. Note that we omit the calculation of the count for space character since it is not part of the JavaScript code contents. The obfuscated code typically has low entropy than normal code, since it often contains repeated characters. It has been experimented that the entropy for obfuscated code is usually lower than 1.2, while the standard code has entropy from 1.2–2.1 [16]. The obfuscated code of s_1 has entropy of 1.1.

Byte occurrence frequency of specific character†. As obfuscated malicious code usually uses customized encoding, and it tends to use excessively specific characters. We use 1-gram model among n-gram model, to examine the occurrence frequency of characters (excluding the space character), which is equivalent to byte occurrence frequency. The byte occurrence is then divided by total characters for normalization. For example, in the obfuscated code in Figure 3(a), the comma character represents the most frequently used character (total of 232 characters), which represents a quarter of the total characters.

```
<!--
eval(String.fromCharCode(118,97,114,32,120,101,119,61,52,53,51,
56,48,48,53,52,51,59,118,97,114,32,103,104,103,52,53,61,34,110,
117,111,116,34,59,118,97,114,32,119,61,34,111,34,59,118,97,114,
32,114,101,54,61,34,108,108,46,34,59,118,97,114,32,104,50,104,
61,34,99,111,109,34,59,118,97,114,32,97,61,34,105,102,114,34,59,
118,97,114,32,115,61,34,104,116,116,34,59,100,111,99,117,109,
101,110,116,46,119,114,105,116,101,40,39,60,39,43,97,43,39,97,
109,101,32,115,114,39,43,39,99,61,34,39,43,115,43,39,112,58,47,
47,39,43,103,104,103,52,53,43,39,39,43,119,43,39,39,43,114,101,
54,43,39,39,43,104,50,104,43,39,47,39,43,39,34,32,119,105,100,
39,43,39,116,104,61,34,49,34,32,104,39,43,39,101,105,103,104,
116,61,34,51,34,62,60,47,105,102,39,43,39,114,97,109,101,62,39,
41,59,32,118,97,114,32,106,104,114,52,61,52,51,50,52,50,50,52));
//-->
```

(a) The original obfuscated version

```
var xew = 453800543;
var ghg45 = "nuot";
var w = "o";
var re6 = "ll.";
var h2h = "com";
var a = "ifr";
var s = "htt";
document.write("<"+a+"ame sr"+"c=\""+s+"p://"+ghg45+""+w+""+re6
+""+h2h+"/"+"\"wid"+"th=\"1\"h"+"eight=\"3\"></if"+"rame>");
var jhr4 = 4324224;
```

(b) The HtmlUnit unpacked version

Figure 3: A malicious sample s_1 of attack Type I

Commenting style†. There are two kinds of comment annotations in JavaScript. "//" is for single-line comments and "<!--...-->" for multiple-line or block comments. A basic evasion technique involves the mixture of these two kinds of comments. For example, the code in Figure 3(a) uses a pair of tags <!-- and //-->, which is expected to be commented out for execution, nevertheless, it is executed due to the fault-tolerance of the browser. The value of this feature records the occurrence frequency of such commenting style, e.g., there is one occurrence for the code in Figure 3(a).

4.2 Inner-Script Program Analysis

A web-page contains multiple JavaScript snippets, some are embedded (internal scripts), while some others are links to other JavaScript fies (external scripts). Analysis of the internal scripts are inner-script analysis, which is considered heavy weight for analysis of all JavaScript snippets. We delay the analysis of external scripts for inter-script analysis (see Section 4.3).

For the unpacked code in Figure 3(b), features about the program information are extracted to characterize the malicious behaviors. The program information includes HTML properties and Document Object Model (DOM) operations, the API usage pattern (i.e., the pattern of function calls) and AST-based features.

Function calls with security risks∗. Malicious JavaScript are usually accomplished by means of code generation, redirecting and DOM operations at runtime. In Figure 4, we list the functions that are with security risks by allowing dynamic code generation, loading or redirecting. Nevertheless, the functions are not used alone but together with some other assistant string operation functions (e.g., concat(), charCodeAt()) that can hide the malicious intention by encoding or encryption. We record the number of occurrences of a function call as a numeric feature, e.g., feature String.fromCharCode() and feature eval() in Figure 3(b) both have a value 1, as they are called only once.

AST features∗. We also consider hierarchical features extracted from the ASTs of unpacked code, e.g. the depth of the AST, the difference between the deepest depth of its subtrees and the shortest depth of its subtrees, the maximum

Function Name	Function Type	Possible Threats
eval() window.setInterval() window.setTimeout()	Dynamic code execution	Dynamic code generation
location.replace() location.assign()	Change current URL	Redirect to malicious URL
getUserAgent() getAppName()	Check browser	Target specific browser
getCookie() setCookie()	Cookie access	Manipulate Cookie
document.addEventListener() element.addEventListener()	Intercepting Events	Block user's operation or emulating
document.write() element.changeAttribute() document.writeln() element.innerHTML() element.insertBefore() element.replaceChild() element.appendChild()	DOM operation	Embed malicious script, invisible java applets, invisible iframe, invisible silverlight, etc.
String.charAt() String.charCodeAt() String.fromCharCode() String.indexOf() String.split()	String operation	Hide intension, by encoding and encryption

Figure 4: Functions with security risks

breadth and the versions of the AST during the generation of the final unpacked code.

There exists many similar assignments and function calls in the ASTs of malicious scripts—this reason leads to a large value of AST depth and maximum breadth. Furthermore, malicious scripts also use multiple levels of dynamic code generation like eval(), i.e., the parameter of one eval function call contains some other function call(s) like eval or document.write. During the process of unpacking these functions that support dynamic code generation, each time one such function like eval is parsed and interpreted, a new version of AST is built. Thus, by analysing the versions of ASTs, we can calculate the times of doing unpacking for functions that support dynamic code generation and loading in a script. Finally, the code in Figure 3(b) has an AST with a depth of 21, a width of 20 and a version number of 5.

Function call patterns[‡]. Different malicious attacks take advantage of distinct vulnerability and have their own behavior models. As API usage (or functional call) pattern is usually adopted to model program behavior, function call patterns can serve as the features predicating a possible type of attack. For the different attack types of samples in the training set, we apply frequent itemset mining on the function calls extracted from ASTs. A mined frequent itemset is a set of function calls that appear together with a support probability greater than a certain threshold. Specifically, we separately apply frequent itemset mining for samples of each attack type, and record all the maximum frequent itemsets with support greater than 20% for each attack type. Finally, 10 mined patterns are manually confirmed as valid function call patterns for the eight attack types.

In Figure 5, we show an example of function call patterns found common in Figure 3(b) for type I attack. As the function newActiveXObject() and createXMLHTTPRequest() are widely used by malware targeting vulnerability in ActiveX components, this pattern captures behaviors of such attack.

4.3 Inter-Script Program Analysis

In inter-script analysis, we count external scripts from other domains. In JSDC, we report malicious JavaScript code at the unit of a code snippet inside <script/>. In our training set, we find some of malicious samples are self-contained in one code snippet of <script/> and do not refer to other scripts to realize malicious intention. However,

Function call pattern:	Intension:
unescape()	obfuscation to evade checking
eval()	dynamic generation
GetCookie()	check Cookie
dateObject.toGMTString()	generate time String used in cookie
SetCookie()	set cookie to mark
document.write()	generate dynamic document content
document.createElement()	create new document element
element.appendChild()	append new element to current one
newActiveXObject()	create new Active object
createXMLHTTPRequest()	download exploit file to local system

Figure 5: An exemplar function call pattern from mining frequent itemsets in function calls

```
<script src="http://xxx.xxx.xxx/a.js"></script>
<script>
 new_element=document.createElement("script");
 new_element.setAttribute("type","text/javascript");
 new_element.setAttribute("src","a.js");//
 document.body.appendChild(new_element);
 function b() {
   a(); //a() is a function in a.js that contains malicious code
 }
</script>
```

Figure 6: A malicious sample s_2 that involves multiple JavaScript snippets

some malicious ones do not contain the actual malicious code. Instead, they just exploit the vulnerability and call some other third party scripts to accomplish the attack. For this type of attack that relies on some other scripts, we propose to adopt the inter-script program analysis.

We extract the times of referring to external scripts as numeric features. Besides, times of external function calls or variable access are also recorded as different numeric features. Our extension of HtmlUnit enables to parse and interpret the external scripts referred by the current one. As shown in Figure 6, the malicious attack refers to an external script "a.js", and it adds the "a.js" into its DOM as an element. Meanwhile, the function b is actually calling the malicious function a in "a.js" to launch the attack.

Miscellaneous and derived features[‡]. Given a suspicious script, to characterize the dynamically generated code or external scripts from other domains, we propose some features for attack type classification: feature *changeSRC* counts the number of changing of the *src* attribute (e.g., for <iframe src="..."/> tag); feature *changeSRC* counts the number of invocation of navigator.userAgent; derived statistic feature *domAndDynamicUsageCnt* counts the number of invocation for APIs that change DOM structure or supporting dynamic execution of JavaScript code in Figure 4; feature *dynamicUsageContentLen* stores the length of contents that are passed as arguments to APIs that support dynamic execution of JavaScript; lastly, feature *referenceError* counts the number of failures in its reference of external scripts. The usefulness of these features are discussed in Section 8.

Note that the inter-script program analysis is more computationally costly, compared with inner-script program analysis, due to the increased size of scripts that need to be analyzed. As inter-script analysis is only applicable to feature extraction for classification of detected malware, the overheads in analysing the comparatively small size of detected malicious are still acceptable in our approach.

5. DYNAMIC CONFIRMATION

To capture the attack behaviors of JavaScript malware at runtime, we propose to model and analyze JavaScript program behaviors by focusing on browser-level system calls. As system calls or actions (high level abstraction of similar system calls) are the interactions of a program with its environment (i.e., the browser in our study), it is effective to model program behaviors (including attack behaviors) based on system calls or actions [19]. Further, we propose to use Deterministic Finite Automata (DFA) to model attack behaviors, which is motivated by the work on detecting anomalous program behaviors based on Finite State Automate (FSA) [35]. The transitions of the DFA are actions, i.e., the high-level abstraction of system calls.

In [35], the dynamic analysis approach has been presented to automatically infer an FSA from a set of execution traces of binary executables. In this study, we have the similar idea, but we are modelling the attack behaviors of JavaScript malware based on browser-level system calls. Specifically, we instrument Mozilla Firefox and capture the method calls to Cross Platform Component Object Model (XPCOM) [10] as browser-level system calls. So our implementation is Firefox specific. The work-flow of dynamic confirmation is shown in Figure 7. First, we execute malicious JavaScript samples to get the training set of (both benign and malicious) execution traces. One good thing to web-based malware detection is no need for triggering. Generally, web-based malware are triggered automatically once the web page is loaded. Second, we perform a preprocessing to simplify these traces by removing security irrelevant system calls (those are not related to security or permission issues). For a common system call that happens in all malicious traces of the same attack type, we wrap it as an action. Then, the simplified traces are converted into action sequences. Finally, given all the actions as the alphabetic in a DFA and the existing action sequences as the training set, we adopt an off-line learning algorithm, i.e., regular positive negative inference (RPNI) [30], to infer a DFA from these execution traces. During the off-line learning, human experts (the authors) are involved to give some positive or negative counterexamples to refine the DFA step by step.

To identify the attack type of a possibly malicious script, we collect its execution traces and check them against these learned attack behaviour models in the form of DFA. If any trace is accepted by a certain DFA, the related script is considered as an instance of this attack type. If the traces of a script are not matched with any predefined or inferred behaviour model, JSDC suggests that this script is probably

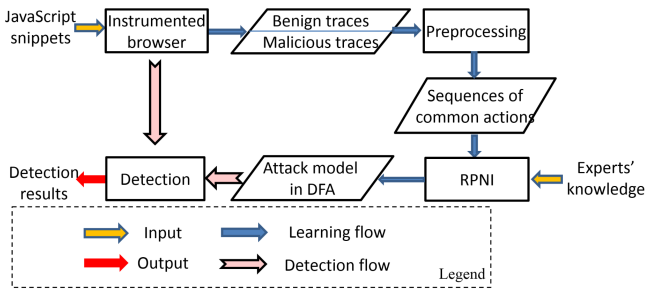

Figure 7: The work-flow of learning and detecting an attack type in dynamic confirmation

System calls	Actions
nsISupportsString.data	
nsICommandLine.resolveURI	$\Rightarrow a$
nsIObserverService.notifyObservers	
nsIWebNavigation.loadURI	$\Rightarrow b$
nsIScriptableUnicodeConverter.convertToByteArray	$\Rightarrow c$
nsICryptoHash.update	$\Rightarrow d$
nsILocalFile.append	$\Rightarrow e$
nsITimer.initWithCallback	$\Rightarrow f$
nsIIOService2.newURI	$\Rightarrow g$
nsIScriptableUnicodeConverter.convertToInputStream	
nsIBoxObject.setProperty	

Figure 8: An execution trace and the actions $\{a, b, c, d, e, f, g\}$ that appear in traces of type I attack

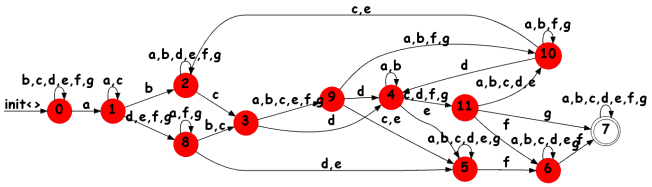

Figure 9: The illustrative DFA of the attack of s_1, which is a part of the actual DFA of type I attack

of a new attack type, or a benign script due to false positive cases introduced in detection.

Given a set of traces from samples of attack type I, we extract the common actions that appear in all malicious traces. Then all traces can be represented by action sequences. For example, the trace in Figure 8 can be represented as an action sequence $\langle a, b, c, d, e, f, g\rangle$. Then with all these action sequences, RPNI is applied to infer a DFA. Further with experts' knowledge, the refined DFA is attained and shown in Figure 9, which generate JavaScript code dynamically and exploit vulnerability of browser to download files into user's local file system. Given the sample s_1 in Figure 3, we can capture traces that are represented as sequences $\langle a, d, e, f, g\rangle$, $\langle a, b, c, d, e, f, g\rangle$ or $\langle a, b, c, b, c, d, e, f, g\rangle$. As these traces are accepted by the learned DFA, s_1 is of type I attack.

Note that for each attack type, we repeat the same work-flow to learn an attack model in DFA. Totally, the eight learned models are available from the JSDC web-site [9]. To identify the possible attack type, the trace of a malicious script needs to be iteratively checked against each of these 8 DFAs, until one matched is found or all are not matched. The system call based behavior modelling is resistant to obfuscation owing to the dynamic analysis, and applicable to malware variants as it captures the essential behaviors of the same attack.

6. IMPLEMENTATION

In this section, we present our implementation details.

Crawler. We use the Heritrix public domain crawler [7] to crawl over 21,000 Internet web sites. Heritrix is a open-source, extensible, web-scale, archival-quality web crawler project. Heritrix is designed around the concepts of profiles and jobs [21]. A profile is a configuration of web crawler. A job is inherited from a profile and can override configuration options. Jobs are queued for processing and will be picked up by an idle crawler thread. Heritrix also supports custom workflows.

HtmlUnit Extension to Obtain Unpacked Code. Analyzing JavaScript with HtmlUnit, we are able to get the un-

packed code (without obfuscation) prior to the actual time-consuming rendering. HtmlUnit is configured so that it does not try to resolve external dependencies (e.g., loading external JavaScript files or third party plugins) referred in the analysed JavaScript. Such a configuration is to make sure that our approach is scalable and light-weighted, since the time taken for the analysis could be quite long if we ensure all dependencies are fulfilled. Nevertheless, this is not a restrictive constraint, since with our experience, most JavaScript malware is designed to be standalone to maximize the chance for successful attack. In some rare events where the external dependencies affect the analysis of HtmlUnit, then our extension falls back to the plain syntactic analysis.

AST Generation and Functional Call Pattern Extraction. To build the AST of unpacked execution-ready code, We use Mozzila Rhino 1.7 [6]—an open source JavaScript engine. The reason to choose Rhino is due to its fault-tolerance and performance in parsing JavaScript code. Rhino is also integrated in HtmlUnit to build AST for the unpacked code that is ready for execution and rendering. We traverse the AST and extract all the function call nodes. We apply the frequent item-set mining implemented in WEKA, a Java-based open source machine learning toolkit [23], to mine function call patterns.

Classifiers. In [27], four types of classifiers, namely ADTree, Random Forest (RF), J48 and Näive Bayes (NB), have been used to solve the malware detection problem, which is a binary classification problem (malicious or benign). In the second phase of our approach, we are doing multi-class classification, to classify the malicious JavaScript code into eight attack types. Thus, we reuse the three classifiers that support multi-class classification, namely RF, J48 and NB. Besides, we also adopt Random Tree (RT).

All of these classifiers are available in the recent version of WEKA. We also use default parameters set-up for these classifiers. The detailed introduction of these classifiers is out of the scope of this paper. We provide a brief summary of each classifier as follows:

- RT: A classifier that build a decision tree with K randomly chosen attributes at each node. It supports estimation of class probabilities based on a hold-out set.
- RF: A non-probabilistic classifier that uses multiple decision trees for training, and predicting the classes by checking the most votes over all decision trees.
- J48: A non-probabilistic classifier that uses single decision tree for classification.
- NB: A probabilistic classifier based on Bayes' theorem with a strong (naive) independence assumptions between the features.

To address the noise due to the different ranges of feature values and treat features fairly, WEKA automatically normalizes values of all features via range and z-normalization. To achieve the best accuracy, WEKA also supports automatic feature selection algorithms.

7. EVALUATION

To test the effectiveness and efficiency of JSDC, we train 4 different classifiers mentioned in previous section with the training data set and perform predication for the testing data set. Particularly, controlled experiments on 20942 labelled scripts are conducted to evaluate the accuracy and run-time performance of JSDC. In addition to the controlled experiments, to exhibit the practical effectiveness, we also apply our approach on 1,400,000 unlabelled scripts to find wild malicious ones contained in public web sites.

Table 1: Data sets used in controlled experiments

Benign data set	**#samples**
Alexa-top500 websites	20000
Malicious data sets	**#samples**
Attack targeting browser vulnerabilities (type I)	150
Browser hijacking attack (type II)	28
Attack targeting Flash (type III)	81
Attack targeting JRE (type IV)	191
Attack based on multimedia (type V)	190
Attack targeting PDF reader (type VI)	101
Malicious redirecting attack (type VII)	92
Attack based on Web attack toolkits (type VIII)	109
Total	942

7.1 Data Set and Experiment Setup

The labelled data sets include 20000 benign samples and 942 malicious ones of 8 attack types, as shown in Table 1. Our malicious data sets originate from three sources. To include the existing common samples, we collect 200 samples of 8 attack types from well-known malware repositories VX-Heaven [4] and another 200 samples from OpenMalware [3]. To include the emerging and new attacks, we manually collect 542 samples from malicious websites reported by Web Inspector [5]. We test these total 942 malicious samples with the on-line service of 54 anti-virus tools provided on VirusTotal [11]. The results report that they are malicious. For the benign data set, we crawl 20,000 scripts from the Alexa-top500 websites[1], from each of which 40 scripts are randomly crawled. We scan them with two anti-virus tools Avast! (version 2014.9.0.2021) and TrendMicro (version 10.4) to ensure that they are benign.

The unlabelled data sets include 1,400,000 scripts that are crawled by Heritrix with randomly selected seeds. Here, URLs in the list that a Web crawler starts with are called the seeds. Our seeds include general web sites of universities, governments, companies, discussion forums, etc. The training samples, testing samples and newly found malware in our study can be found from the JSDC web-site [9].

Our experimental environment is a Dell optiplex 990 PC (Intel Core i7-2600 3.40GHz, 8G memory), running Windows 7 64-bit Enterprise.

7.2 Evaluation of Malware Detection

7.2.1 *Detection on the Labelled Data Sets*

To test the accuracy of the classifiers for the labelled data sets, malicious and benign scripts are all mixed and then separated into multiple partitions. To avoid possible errors introduced in random partition of training set and testing set, we do not split all samples into a training partition (75%) and a predication partition (25%). Instead, we adopt 10-fold cross-validation (10-fold CV) strategy to train the 4 classifiers: RF, J48, NB and RT.

Accuracy of the trained classifiers. Table 2 shows the accuracies of trained classifiers on the labelled data sets. The accuracy is calculated by measuring the ratio of the number of successful classifications to the total number of samples. As there are more than 40 different features for classifica-

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

Table 2: The accuracies of different classifiers

ML classifier	Accuracy	FP rate	FN rate
RandomForest (RF)	99.9522%	0.2123%	0.8492%
J48	99.8615%	0.7431%	2.335%
Näive Bayes (NB)	98.2237%	1.127%	4.5280%
RandomTree (RT)	99.8758%	0.3609%	1.4862%

```
<!--
if(navigator.userAgent.match(/(android|midp|j2me|symbian|
series 60|symbos|windowsmobile|windowsce|ppc|smartphone|
blackberry|mtk|bada|windows phone)/i)!==null){
window.location = "http://mirolend.h19.ru/away.php?sid=5";
} //-->
```

Figure 10: A code example of FN cases

tion of malicious and benign scripts, we use the function of automated searching and selection of predicate features in WEKA. Thus, as Table 2 shows, all the four classifiers have achieved an overall accuracy above 98%. Note that the overall accuracy is probably dominated by the large number of correct classification of benign sample. Thus, checking the specific false positive and false negative cases will provide more insightful observations.

In Table 2, the worst classifier is Näive Bayes, whose FN rate of malicious samples is 4.5280%. However, due to the overwhelming percentage of benign samples, a high FN rate (4.5%) of malicious samples does not affect the overall accuracy significantly. The other three classifiers have low FP rate. One of the FN cases for RF, J48 and RT is shown in Figure 10. This sample belongs to malicious redirects (type VII) and requires the client side to be a mobile device. Thus, HtmlUnit cannot parse and unpack the code properly so that the extracted feature vector is not accurate. Overall, the FN of malicious samples range from 0.3500% to 4.5280% for different classifiers. Thus, our tool achieves a detection ratio 99.65% (the RF classifer) on the whole labelled malicious samples. It is reasonable to get such high detection ratio, as in 10-fold CV the different folders (partitions) are mutually used for training and predication for each other.

Comparison with other tools. To compare the accuracy and performance of our approach with the state-of-the-art tools, we consider research prototype JSand[2] [13] and other 11 popular anti-virus tools. The 11 anti-virus tools include the open-source anti-virus software ClamAV[3] and the 2014 best reviewed ten anti-virus products[4]:—AVG, Avast!, Bitdefender, F-Secure, Gdata, Kaspersky, McAfee, Panda, Symantec and TrendMicro. JSand and ClamAV are used in the previous study for comparison [33]. JSand is available as on-line service from this link: https://wepawet.iseclab.org/index.php. ClamAV and the best ten anti-virus tools are available as on-line service from VirusTotal [11]. To enable the latest JSand 2.6 for detection, we upload 942 malicious samples and analyse them for five times to avoid the cached result provided by JSand 1.3.

The results are summarized in Table 3. For each malicious sample we also count the number of tools, which succeed in detection, among the total 54 tools provided on VirusTotal. On average, each malicious sample is reported by 21 tools. Given any malicious sample in our labelled data sets, usually 6 to 7 among the best reviewed ten anti-virus tools can detect. These tools' detection ratio ranges from 5% to 93%. Note that we are not comparing these tools to find the best ones, but investigating the limitations of mainstream anti-virus tools. Our goal is to find the malicious scripts missed by JSand and the best reviewed ten anti-virus tools. The best classifier RF in Table 2 can detect 939 out of 942 (99.68%) malicious samples. The failed cases are because of some implementation issues relating to our HtmlUnit-based infrastruction, which can not emulate all versions of mainstream browsers.

[2]JSand supports three types of results, i.e., benign, suspicious and malicious. We consider it is detected if JSand reports suspicious or malicious.

[3]ClamAV: http://www.clamav.net/

[4]2014 Best Antivirus Software Review: http://anti-virus-software-review.toptenreviews.com/

Table 3: Detection Ratio of our approach and other tools on labelled malicious data sets

Tool	Detection %	Tool	Detection %
JSDC	99.68%	McAfee	59.87%
Avast!	93.84%	Symantec	27.28%
Kaspersky	86.31%	JSAND	25.58%
Gdata	85.88%	Trend	22.08%
Bitdefender	83.23%	ClamAV	9.24%
F-Secure	82.38%	Panda	5.31%
AVG	76.22%		

7.2.2 *Detection on the Unlabelled Data Set*

After we get the trained classifiers from the labelled data sets, we apply the learned model to unlabelled data sets for predication. Among these 1,400,000 wildly crawled scripts, the best trained classifier RF predicates 1,530 snippets as malicious.

To verify our prediction accuracy, we randomly select 100 from 1,530 reported unique samples. We manually inspect these 100 cases and find only 1 FP case in these 100 samples. For the rest 99 TP cases, we test them by other tools mentioned in Section 7.2.1. We find 11 out of 99 TP cases are missed by all the tools. For example, we found an exploit to CVE-2014-1580 is missed by all the tools. Mozilla Firefox before 33.0 does not properly initialize memory for GIF images, which allows remote attackers to obtain sensitive information from process memory via a crafted web page that triggers a sequence of rendering operations for truncated GIF data within a CANVAS element. The results in Table 4 show individual detection ratio for each tool. The best tool can achieve a detection ratio of 48.49%. We observe two facts: first, these tools perform much worse on the unlabelled samples than the labelled ones. Most of the labelled ones are from public repositories and may have been scanned by these tools, but unlabelled ones are new merging variants. Second, the ranking of detection ratio of these tools in Table 3 is generally consistent with the ranking in Table 4, although labelled and unlabelled samples are from different origins. The above observations imply that detection of wild and emerging malware is not easy. Note that our comparison mainly focuses on detection ratio (or FN ratio) rather than FP ratio, since existing studies [18, 33] have found that existing JavaScript malware detection tools have low FP but high FN.

7.2.3 *Performance of Malware Detection*

To know the potential of our approach being an off-line large-scale anti-virus tool or an on-line real-time scanner as a browser plugin, we investigate the performance of each step in malware detection.

Table 4: Detection ratio of other tools on the 99 unique samples reported by our approach

Tool	Detection%	Tool	Detection%
Kaspersky	48.49%	McAfee	22.22%
Avast!	46.47%	JSAND	10.10%
Gdata	34.34%	Trend	6.06%
Bitdefender	30.30%	Symantec	3.03%
F-Secure	30.30%	ClamAV	2.02%
AVG	27.27%	Panda	0.00%

Table 5: The running time for different classifiers

Operation	Num	Time(s)	Avg(ms)
Feature extraction	20942	1660.7	79.3
Training(RandomForest)	20942	0.785	0.037
Training(J48)	20942	0.364	0.017
Training(Näive Bayes)	20942	0.124	0.006
Training(RandomTree)	20942	0.275	0.013
Detection(RandomForest)	1,400,000	57.4	0.041
Detection(J48)	1,400,000	26.6	0.019
Detection(Näive Bayes)	1,400,000	8.4	0.006
Detection(RandomTree)	1,400,000	19.6	0.014

As shown in Table 5, we list the time used for different classifiers when the whole labelled data sets serve as the training set. The main overheads come from the feature extraction step. Although the total time of feature extraction for 20942 scripts in training sets is 1660.7 seconds, the average time for each script is just 79.3 ms. As mentioned in Section 4.3, feature extraction for effective inner and inter program analysis requires the execution ready code that is after several times of unpacking. Thus, unpacking code, generating ASTs and doing program analysis are the major time-consuming tasks at the step of feature extraction. However, feature extraction can be processed for only once, and the resulting feature vectors can be stored in database for future usage. Table 5 also lists the total training and detection time for different classifiers. For 20942 feature vectors, the training of classifiers only need less than 1 seconds. We can also observe that Näive Bayes classier, as the worst classifier in Table 2, exhibits the best performance—with only 0.006 ms needed for each script in detection stage. The sum of average feature extraction time and detection time approximately provides an estimate of the scanning time for a single script. Given the ignorable average detection time for each script, the average feature extraction time (79.3 ms) suggests that our approach can be used as a browser plug-in for real-time detection of malicious JavaScript.

To further evaluate the potential of our tool being an offline large-scale detector, we investigate the detection time needed for a large set of unlabelled scripts. Apart from feature extraction time, the detection only needs up to 1-2 minutes to classify 1.5 million scripts. In the real application of our approach for detection, we use Heritrix to crawl the pages and, meanwhile, HtmlUnit to unpack code and extract feature vectors. Thus, crawling and feature extraction are processed in parallel. The detection can be accomplished with different classifiers and different selected features.

7.3 Evaluation of Attack Type Classification

We measure the accuracy and performance of the attack type classification on the labelled and unlabelled samples. We further investigate when the dynamic confirmation is needed and how it can improve the overall accuracy.

7.3.1 Accuracy of the Trained Classifiers

Table 6: The confusion matrix of using RandomForest on the 942 labelled malicious samples

a	b	c	d	e	f	g	h	<–classified as
139	0	0	0	9	2	0	0	a = type I
0	23	4	0	0	0	0	1	b = type II
1	1	74	1	0	0	1	3	c = type III
0	0	2	179	9	0	1	0	d = type IV
1	0	0	0	179	10	0	0	e = type V
0	0	0	0	19	82	0	0	f = type VI
0	0	0	1	1	0	87	3	g = type VII
0	0	0	1	0	0	3	105	h = type VIII

Table 7: Attack classification on the 1530 malicious ones detected from 1,400,000 unlabelled samples

Type	type I	type II	type III	type IV
Num	113 (7.39%)	10 (0.65%)	75 (4.90%)	253 (16.54%)
Type	type V	type VI	type VII	type VIII
Num	202 (13.20%)	101 (6.60%)	350 (22.88%)	426 (27.84%)

We adopt 10-folder CV strategy to train the 4 classifiers (RF, J48, NB and RT) with the 942 labelled samples in Table 1. The accuracy of these 4 classifiers is 92.14% (RF), 90.22% (J48), 83.44% (NB) and 90.13% (RT), respectively. Table 6 shows how the 942 samples are classified into the 8 attack types according to RF. The sum of the entries on the matrix's main diagonal, namely 868 (92.1444%), are correctly classified samples. Among the 74 wrongly classified samples, 19 samples of type VI are wrongly classified as type V, while 10 of type V are wrongly classified as type VI. The direct reason is that some samples of type V and type VI are quite similar—their feature values show no significant differences. For the other 45 out of 74 wrongly classified cases, we find that most of them fall into the grey zone without a dominant certainty value for any attack type. The other three classifiers achieve slight worse accuracies. We also observe that they have similar problems in distinguishing type V and type VI, and classifying a number of uncertain cases. Thus, the trained classifiers can be ready for practical usage if we can address the above two problems via combining dynamic attack confirmation (see Section 7.4).

7.3.2 Predication on the Unlabelled Data Set

In Section 7.2.2, 1530 malicious samples are detected by our approach from 1.4 million unlabelled scripts. We apply the best trained classifiers, namely RF, to classify these samples into different attack types. The classification output is listed in Table 7. To measure the accuracy, we manually inspect 10% of samples of each type (for type II and type III that has few samples we randomly choose 10 samples). Totally, we check 164 samples and confirm that they are indeed malicious. On these 164 samples, the trained RF classifier achieves an accuracy of 87.8%, which means correctly classifying 144 out of 164 samples. Among the 20 mistakenly classified ones, 9 samples do not belong to any of the eight attack types and 11 samples are classified into the wrong types. We observe that 3 wrong cases are related to type V or VI confusion, and 9 cases fall into the grey zone. Thus, the sole supervised classification cannot accurately predicate the attack type of JavaScript malware, since feature values extracted by mostly-static analysis may not characterize the actual attack behaviors. Especially, for those cases that fall into grey zones, we need to apply dynamic confirmation to to analyse them by execution.

Table 8: The certainty value and the number of samples that fall into grey zone

certainty	certain#	total	uncertain#	uncertain %
1	764	1530	766	51.31%
⩾0.9	854	1530	676	50.07%
⩾0.8	994	1530	536	44.18%
⩾0.7	1296	1530	234	15.29%
⩾0.6	1311	1530	219	14.31%

7.4 Combining Dynamic Confirmation and Machine Learning Classification

The dynamic confirmation is applied on uncertain cases that fall into the grey zone during attack type classification.

To effectively select the uncertain cases, we investigate how the probabilities of the predicated class are distributed for the 1530 samples that are classified by RF in Section 7.3.2. In Table 8, we define the *certainty* as the dominant one among a sample's probabilities to be eight attack types. For example, $certainty = 1$ means there exists one dominant probability (100%) for a sample to be the related predicated attack type. *certain#* refers to the number of samples with this certainty value, i.e., 764 out of 1530 for $certainty = 1$. Thus, the number of uncertain cases without a dominant probability (*uncertain#*) is 766, which comprises 51.31% of the total 1530 samples.

Accuracy of Dynamic Attack Confirmation. From Table 8, we observe that 234 samples are without a dominant probability larger than 0.7 (234 samples are in the grey zone if $certainty = 0.7$). We apply the dynamic confirmation on these 234 samples. As described in Section 5, we learn a DFA for each attack type from the related ones in the training set of 942 samples. We separately execute the 234 samples and get 10 traces for each of them. Then these traces are used for acceptance check on the learned DFAs. If any trace is accepted, the corresponding sample is matched with the DFA of the compared attack type. Finally, we compare the matching results with manual verification results to measure the accuracy. Totally, the dynamic confirmation correctly classifies 220 samples out of 234, achieving an accuracy of 94.02%. Among eight attack types, type I shows the best accuracy (95%) for dynamic confirmation while type VI shows the worst (90.91%).

Thus, dynamic confirmation alone can achieve good accuracy for attack type classification. However, the problem is that it is not scalable. It takes about 1,544 seconds to analyse the 234 samples, i.e., averagely 6.6 seconds for each. Among 234 samples without a certainty value larger than 0.7, RF achieves an accuracy of 67.09%. Thus, among 234 grey zone samples, dynamic confirmation can rightly classify 63 more than RF.

Performance of Attack Type Classification.

As shown in Table 9, the training step of attack type classification takes about 0.01-0.12 seconds, which does not include the time of feature extraction. The predication step takes only about 0.001-0.05 seconds. Thus, the performance of these 4 classifiers is quite good, owing to the comparatively small number of samples to be handled in attack type classification. Besides, these classifiers also exhibit consistent performance — that is a classier fast in malware detection is also fast in attack type classification. Note that the time for feature extraction in this step is the same as that in malware detection step, as all feature values are extracted at once. Feature extraction averagely takes 0.1 second for each of 1530 detected malware.

Table 9: Running time of different classifiers in type classification

Operation	Num	Time(s)	Avg(ms)
Training(RandomForest)	942	0.1	0.11
Training(J48)	942	0.12	0.13
Training(Näive Bayes)	942	0.04	0.04
Training(RandomTree)	942	0.01	0.01
Detection(RandomForest)	1530	0.05	0.0033
Detection(J48)	1530	0.01	0.0006
Detection(Näive Bayes)	1530	0.001	0.00006
Detection(RandomTree)	1530	0.02	0.001

With regard to the time of dynamic confirmation on 234 samples, it takes 1,544 seconds. Including the 168 seconds used for classification (with feature extraction time for 1530 samples), it takes 1,712 seconds to finish identifying attack types for the 1530 malware, i.e., around 1 second per malware. Thus, combining this two techniques can improve the overall accuracy and meanwhile attain good performance.

8. DISCUSSION

Two- or one-phase machine learning classification? A concern is that how the accuracy will be if we make two phase classification into one—that is we do classification at once according to samples with benign labels and labels of eight attack types. On the 20942 training samples, the accuracy of the 4 trained classifiers is 90.99% (RF), 85.74% (J48), 77.15% (NB) and 88.27% (RT), respectively. We examine the results and find the difference for RF is not significant. Among 20000 benign samples, only one is wrongly classified for RF, while among 942 malicious samples, 84 are wrongly classified for RF. In contrast, 74 out of 942 are wrongly classified in our two phase classification of RF. But NB shows significant differences—it wrongly classifies 203 out of 942 malicious samples in one phase classification, in contrast, this number is 156 in our two phase classification. As the training time is almost ignorable if feature extraction is done, we consider two phase classification is worth.

Predicative features. We find that the malware detection has some decisive features: e.g., 98% of benign scripts use eval function only 0 or 1 times, while 87% malicious ones use more than 2 times, and as many as 1778 times. During attack type classification, we observe that there are no decisive features that can exclusively characterize a certain attack type. Instead, we indeed find some predicative features that can be effective in most cases. Specifically, among our 942 training malicious samples, we observe the following facts: 89% of Type I samples have feature $changeSRC < 1$; 52% of Type II have feature *element.appendChild* in Figure 4 with a value>20 ; 74% of Type III have feature *eval* in Figure 4 with a value>1000; 83% of Type IV have feature $GetUserAgent > 2.5$; 87% of Type V have feature $referenceError < 1$; 74% of Type VI have feature $invisibleIframe$=1, which means the frame is invisible; 79% of Type VII have feature $domAndDynamicUsageCnt$ <1; 67% of Type VIII have feature $dynamicUsaeContentLen$ >5000.

Thread to validity. First, the training sets are from three origins: VXHeaven, OpenMalware and Web Inspector. Thus, the collected samples directly affect the accuracy of trained classifiers. To address this problem, we need to further investigate the impact of sample size and rep-

resentativeness on the results. Second, the parameters and set-up for machine learning classifiers are the default ones. Further investigation should be conducted to see the effects of parameters. Lastly, the dynamic attack confirmation relies on experts' knowledge to refine the learned DFA. The rationale of learned DFAs can affect the accuracy of attack confirmation—checking if a trace of a sample is accepted by the DFA of a certain attack type.

9. RELATED WORK

Since the past few years, JavaScript malware detection has intrigued the interest of security researchers and many approaches have been proposed. Existing approaches to JavaScript malware detection mostly rely on static analysis with machine learning techniques or dynamic analysis with behavioural abnormality checking. However, beyond the mere detection, there are few studies that focus on malware classification according to vulnerability or attack behaviors.

9.1 Machine Learning Approaches

In 2009, Likarish *et al.* [27] adopted 4 classifiers (i.e., NB, ADTree, SVM and RIPPER) to detect obfuscated JavaScript malware. The assumption of their approach is that malware utilizes obfuscation to hide the exploits and to evade the detection. They reported that features extracted from the strings of obfuscated scripts can distinguish obfuscated (malicious) scripts from non-obfuscated (benign) ones by machine learning classifiers. Likarish *et al.* acknowledged that the false positive rate is due to obfuscated but benign scripts, which poses threat to the assumption. However, in our study, we use not only features on obfuscation, but also other features on program information. By this, we can reduce false positive rate due to obfuscated but benign scripts.

In 2010, JSand (JavaScript Anomaly-based aNalysis and Detection) [17] was presented to detect Drive-by Downloads (DbD) attacks. JSand identifies 10 features characterizing intrinsic events of a drive-by download attack, and then uses these features for machine learning technique (NB) to detect malicious samples. These 10 features are extracted from 4 aspects (redirection, de-obfuscation, environmental context and exploitation) by dynamic anomaly detection with emulation in HtmlUnit [1]. Different from our work, JSand adopts dynamic analysis and fails to investigate the impact of the different classifiers. Dynamic anomaly-based analysis can be effective but not efficient—JSand needs 2:22 hours to scan the *Wepawet-bad dataset* which consists of 531 URLs.

Also in 2010, Cujo [33] uses hybrid analysis to on-the-fly capture program information (by static analysis) and execution traces (by dynamic analysis) of JavaScript program. The authors explored the distinct features for heap-spraying attacks and obfuscated attacks. But they failed to examine features for obfuscated but benign scripts, and overlooked classification according to vulnerabilities or attack behaviors as we do in this study. Note that Cujo requires executing every script to capture dynamic features, while our approach only executes scripts that fall into grey zone for dynamic confirmation. Finally, Cujo takes averagely 500ms to analyze a webpage. In contrast, JSDC takes about 80 ms in detection per script, including 70 ms for feature extraction and 10 ms for classification.

Later in 2011, Canali *et al.* [14] proposed a filter, called Prophiler, to quickly filter out benign pages and forward to the costly analysis tools only the suspicious pages that are highly malicious. The filter's detection models are learned based on 70 features extracted from HTML contents of a page, from the associated JavaScript code, and from the corresponding URL. Then, multiple classifiers, Bayes Net(BN), J48, Logistic, RT and RF, are used to classify the malicious and benign web-pages. For JavaScript, the FP of these classifiers is good, ranging from 0.0% to 1.7%, but the FN ranges from 18.1% to 81%, which is worser than our approach.

In the same year, AST is used to extract characteristics for malicious JavaScript detection [18]. Curtsinger *et al.* [18] presented the tool Zozzle that predicates the benignity or maliciousness of JavaScript by leveraging features associated with AST contextual information. Around one to two thousand of features are extracted from the hierarchical structure and texts in ASTs, and then the classifier NB is applied. To remove the noise due to obfuscation, Zozzle uses Detours binary instrumentation to get the final, unpacked code for accurate AST generation. After tuning up with different set-up, the authors reported the FP ranges from 0.0003% to 4.5%, and the FN from 1.26% to 11.08%.

To sum up, none of the existing studies investigate the effective features for classifying JavaScript malware into various attack types as we do in this study. Additionally, we propose to apply dynamic confirmation to the cases that fall into grey zone in classification. By this, we can improve the overall accuracy and meanwhile achieve scalability.

9.2 Other Detection Techniques

Various approaches have been presented to detect specialized JavaScript malware in line with different attacks.

Attacks of obfuscated malware. In [16], the parameters for dangerous functions (eval, document.write, and so on) in web pages are processed to get 3 types of metrics data, i.e., N-gram, entropy and word size. The metrics data is compared with a certain threshold that is learned from empirical study to judge the maliciousness of the corresponding web page. Recently, JStill [38] detects obfuscated attacks based on the observation that—JavaScript has to be de-obfuscated before doing its malicious intention, but the deobfuscation process inevitably invokes certain functions.

Policy violation attacks. As early as in 2005, Hallaraker *et al.* [24] monitored the JavsScript execution and then checked the execution trace against the high-level security policies (e.g., the *same-origin* policy and the *signed-script* policy), to detect malicious attack behaviour.

Heap spread attacks. In 2009, Egele *et al.* [20] managed to detect heap spread attacks by checking instantiation of shell-code strings in the heap—specifically—checking if the string contains executable x86 code. The similar idea is adopted in Nozzle [32], which individually examines all objects in the heap, interpreting object content as code and conducting a static analysis on the code to discover exploits.

JavaScript-bearing PDF documents. Tzermias *et al.* [36] presents MDScan, which instrumented Firefox SpiderMonkey. In such way, all the allocated string objects are scanned by the shellcode detector Nemu [31], and possible exploits are detected. MDScan failed to address this problem completely since it only searched for legitimate JavaScript inclusions (denoted by the /JS tag), while an attacker can easily circumvent this by encoding the script as text and evaluate it at runtime (e.g. via eval).

JavaScript worms. The tool Spectator [28] detects a worm whenever a unusually long propagation chain is found. The idea is using a web proxy to monitor the cookie session and propagation chain for identification of worm replication.

In [15], PathCutter adopts the idea of running an individual script in different isolated and trusted components of web-pages. It prevents the propagation of worms by granting scripts the permission to do only legitimate operations.

10. CONCLUSION

A key to protecting user from exploitation is the rigorous eliminating of malware on Internet. To this end, we propose a method to accelerate the process of manual malware detection by suggesting potentially malware and their attack types to an analyst or an end-user. Our method not only learned features of maliciousness but also of attack type. Therefore, we can tell not only presence of malware, but malicious behaviors with attack approach information by virtue of dynamic attack confirmation. We also demonstrated our effectiveness and efficiency by empirical wild prediction. Among over 1,400,000 scripts, we find over 1,500 malware with 8 attack types. Our detection speed is scalable with below 80 ms per script. The attack type classification takes around 1 second for each malware, combining machine learning classifiers and dynamic attack confirmation.

11. ACKNOWLEDGEMENTS

This work is supported by "Formal Verification on Cloud" project under Grant No: M4081155.020 and "Vulnerability Detection in Binary Code" project under Grant No: M40 61366.680.

12. REFERENCES

[1] Htmlunit. http://htmlunit.sourceforge.net/.

[2] Internet security threat report. http://www.symantec.com/security_response/publications/threatreport.jsp.

[3] Openmalware. http://http://oc.gtisc.gatech.edu:8080/.

[4] Vxheaven. http://vxheavens.com/.

[5] Web inspector: Website security with malware scan and pci compliance. http://www.webinspector.com/.

[6] Rhino. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino, 2005.

[7] The heritrix web crawler project. http://crawler.archive.org/, 2007.

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

[9] JSDC. https://sites.google.com/site/jsmalwaredetection/models.html, 2014.

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

[11] VirusTotal. https://www.virustotal.com/, 2014.

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

[13] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. Jsand: Complete client-side sandboxing of third-party javascript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 1–10, New York, NY, USA, 2012. ACM.

[14] 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.

[15] 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.

[16] Y. Choi, T. Kim, S. Choi, and C. Lee. Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis. In *International Conference on Future Generation Information Technology*, pages 160–172, Berlin, Heidelberg, Germany, 2009.

[17] 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.

[18] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *the 20th USENIX conference on Security*, Berkeley, CA, USA, 2011.

[19] 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.

[20] 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.

[21] B. Feinstein and D. Peck. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. In *DEFCON 15*, USA, 2007.

[22] J. J. Garrett et al. Ajax: A new approach to web applications. 2005.

[23] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

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

[25] A. Ikinci, T. Holz, and F. C. Freiling. Monkey-spider: Detecting malicious websites with low-interaction honeyclients. In *Sicherheit*, volume 8, pages 407–421, 2008.

[26] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An automated Approach to the Detection of Evasive Web-based Malware. In *USENIX Security*, pages 637–652, 2013.

[27] P. Likarish, E. Jung, and I. Jo. Obfuscated malicious javascript detection using classification techniques. In *MALWARE*, pages 47–54, Montreal, QC, Oct. 2009.

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

[29] G. Mohr, M. Stack, I. Rnitovic, D. Avery, and M. Kimpton. Introduction to heritrix. In *4th International Web Archiving Workshop*, 2004.

[30] J. Oncina and P. Garcia. Inferring Regular Languages in Polynomial Update Time. *Pattern Recognition and Image Analysis*, pages 49–61, 1992.

[31] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. An empirical study of real-world polymorphic code injection attacks. In *LEET*, 2009.

[32] 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.

[33] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *ACSAC*, pages 31–39, 2010.

[34] M. Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.

[35] 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.

[36] 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.

[37] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *NDSS*, 2006.

[38] W. Xu, F. Zhang, and S. Zhu. Jstill: mostly static detection of obfuscated malicious javascript code. In *CODASPY*, pages 117–128, 2013.