Efficient and effective realtime prediction of drive-by download attacks

Gaya K. Jayasinghe[1], J. Shane Culpepper[1], Peter Bertok[1],

Abstract

Drive-by download attacks are common attack vector for compromising personal computers. While several alternatives to mitigate the threat have been proposed, approaches to realtime detection of drive-by download attacks has been predominantly limited to static and semi-dynamic analysis techniques. These techniques examine the original or deobfuscated JavaScript source code to assess the potential maliciousness of a webpage. However, static and semi-dynamic analysis techniques are vulnerable to commonly employed evasion techniques. Dynamic anomaly detection approaches are less susceptible to targeted evasion, but are used less often as a realtime solution on the individual systems because these techniques are typically resource intensive. This paper presents a novel approach to detect drive-by downloads in web browser environments using low resource dynamic analysis. By dynamically monitoring the bytecode stream generated by a web browser during rendering, the approach is able to detect previously unseen drive-by download attacks at runtime. The proposed method is effective, space efficient, and performs the analysis with low performance overhead, making the approach amenable to in-browser drive-by download detection on resource constrained devices, such as mobile phones.

Keywords: Drive-by downloads, web client exploits, anomaly detection, machine learning, dynamic analysis

1. Introduction

Traditional intrusion prevention techniques, such as firewalls, provide protection against a wide variety of *push-based* security exploits. But, our increased reliance on the web for routine activities such as communication and entertainment has created an opportunity for adversaries to infect many client computers simultaneously using *pull-based* methods. As a result, recent web-based exploits are predominantly deployed using pull-based mechanisms [38]. An increasingly common class of pull-based client-side security exploits delivered via the internet are *drive-by downloads* [33, 37]. These attacks usually leverage web browser vulnerabilities in order to hide malicious software downloads onto a computer or mobile device. Backdoors that provide unhindered remote access to the compromised client devices, trojan downloaders that fetch and install other programs from the web, and trojan proxies that re-route Internet traffic are some of the malware installed via drive-by downloads. Consequentially, a user may loose sensitive data, or the client computer may become a part of a large *botnet* [38].

JavaScript is the primary programming language used to implement and deploy context rich web applications. However, JavaScript is not a security conscious language [18, 23]. The ability to run arbitrary code on client devices, and the tightly integrated interaction between browser and supporting technologies makes JavaScript an obvious candidate for targeted drive-by download attacks [9, 50]. In a recent study of common attack vectors using drive-by download attacks, Provos et al. [37] discovered more than 3 million malicious URLs. Adversaries use a variety of tactics to lure users to malicious webpages. Compromised websites, user contributed content, third-party widgets and advertising are commonly used to insert malicious content into benign sites [38]. In fact, finding malicious web content injected into otherwise trusted websites is now more common than stumbling upon rogue websites which are explicitly designed to distribute malware [50]. Poisoning the search results of popular search engines using search engine optimization is another common strategy used to draw users to malicious websites [22, 50].

Email addresses: gaya.jayasinghe@student.rmit.edu.au (Gaya K. Jayasinghe), shane.culpepper@rmit.edu.au (J. Shane Culpepper), peter.bertok@rmit.edu.au (Peter Bertok)

As a consequence, detection and prevention of drive-by downloads has been an important topic to security researchers. Typical solutions may run *off-line* on server infrastructure, producing *malware signatures* and *blacklists* [52], while others provide *realtime* protection on client devices. Analyzing the properties of webpage URLs [30, 51, 54], and investigating the embedded webpage content [4, 5, 7, 42] are two defensive strategies commonly employed to detect malicious webpages. URL based strategies operate on the server-side as an off-line solution to generate blacklists [51, 54]. In contrast, webpage content analysis attempts to detect webpages performing drive-by download attacks on-the-fly. If the content analysis is suitably efficient, the approach may be used for realtime detection on the client-side. *Static* and *dynamic* (emulation based) analysis are two possible approaches to webpage content analysis.

Static analysis in the form of *misuse detection*, which uses a set of pre-defined rules and heuristic signatures to detect malicious patterns, is often used on the client-side in antivirus solutions. Static analysis integrated with machine learning to detect drive-by downloads was subsequently proposed by Rieck et al. [42]. Static analysis can reveal malicious features in JavaScript code, but dynamic features offered by JavaScript language [41] and obfuscated or encrypted code appearing in both benign and malicious pages [25] can marginalize the effectiveness of approaches depending solely on static analysis. To overcome evasion by code obfuscation, a semi-dynamic approach that detect malware using static analysis of deobfuscated code [5] has also been explored. Dynamic analysis techniques in the form of misuse detection [7, 35, 45], and anomaly detection [4, 42] has also been extensively studied. Compared to static analysis, dynamic analysis has the potential to detect malicious webpages more accurately as the attack can be exposed by monitoring the execution sequence.

Anomaly detection techniques commonly use machine learning and classification to identify drive-by download attacks, and differ mainly in feature extraction techniques. The features for anomaly detection can be manually crafted [3, 4], automatically extracted from the training dataset [5], or a combination of both [30, 42]. Automatic feature extraction has the advantage of not requiring domain specific knowledge to come up with a set of features. Existing automatic feature extraction methods for dynamic analysis include sample specific values such as user defined JavaScript identifiers (variable and function names) and values (strings, integers, etc.) to construct the feature space for machine learning [5, 42]. A large portion of today's JavaScript attacks either copy and paste, or borrow code from previous attacks [5]. Capturing these unique identifiers in the feature space can lead to high detection rates. However, inclusion of variable assignments in the automatically generated feature space can also result in a feature space of unbounded size; an *overfitting* (*high variance*) classifier [17, 31]; and/or a resource intensive classification process [17]. The problem is much more severe when automatic feature extraction is employed with dynamic analysis. As a result, dynamic analysis solutions with automatic feature extraction are often computationally expensive, and have not gained widespread usage in resource constrained devices.

A proxy server is a reasonable compromise between signature-based and fully dynamic anomaly detection on the client-side as the approach can protect multiple users simultaneously on a private network, and localize computational costs to a single machine [28, 32, 42]. The proxy-based approach can also be augmented using semi-dynamic solutions that detect malware using static analysis of deobfuscated code [5]. Neither of these solutions is entirely satisfying. While blacklists and signature-based solutions somewhat mitigate the spread of malware, empirical studies show that new outbreaks tend to outpace updates to commercial databases [4, 42]. Static and semi-dynamic anomaly detection approaches can also be evaded using code obfuscation or by rearranging the code [5]. So, developing novel client-side dynamic anomaly detection approaches capable of discovering previously unrecognized exploits without bombarding the user with a disproportionate number of false alarms, and that minimize resource usage remain an attractive research goal.

Our contributions: This research is a first step in designing new dynamic anomaly detection algorithms that support automatic feature extraction, and capable of balancing effectiveness and efficiency with low resource overhead, and rests on following key contributions:

1. Effective, space efficient, and transparent dynamic anomaly detection algorithms, which can be run on resource constrained consumer devices.
2. Automatically generated generic and limited feature space free of sample specific values, which requires no domain specific knowledge of JavaScript malware to operate and maintain.
3. The classifier that can be trained using very few positive training instances, in contrast to existing dynamic approaches using automatic feature extraction.

Line	JavaScript Code

```
1   // (a) Shell code and construction of the NOP sled
2   shellcode = unescape("%u9090%u9090...");
3   heap_block_size = 0x400000;
4   shellcode_size = shellcode.length * 2;
5   spray_slide_size = heap_block_size - (shellcode_size + 0x38);
6   spray_slide = unescape("%u0505%u0505");
7   while (spray_slide.length * 2 < spray_slide_size) {
8    spray_slide += spray_slide;
9   }
10  spray_slide = spray_slide.substring(0, spray_slide_size / 2);
11
12  // (b) Heap spray
13  heap_spray_to_address = 0x05050505;
14  heap_blocks = (heap_spray_to_address - 0x400000) / heap_block_size;
15  memory = new Array();
16  for (i=0; i < heap_blocks; i++) {
17   memory[i] = spray_slide + shellcode;
18  }
19
20  // (c) Exploitation of the browser vulnerability
21  ...
```

Figure 1: JavaScript code segment from a heap spraying attack.

The structure for the remainder of this paper is: A brief overview of JavaScript and drive-by download attacks is presented in Section 2; In Section 3 prior work on detecting and protecting users from drive-by download attacks is reviewed; Our approach is described in Section 4; Section 5 evaluates, compares, and contrasts our method against current state-of-the-art; and the paper is concluded in Section 6.

2. Background

In this section, we compare and contrast current JavaScript drive-by download attacks using a simple example, and describe basic data mining algorithms suitable for detection of drive-by download attacks. Finally, we review the common evaluation metrics used to empirically validate our results.

2.1. JavaScript drive-by download attacks

Figure 1 is a sample snippet of JavaScript code used to perform a drive-by download attack. The code contains a *shellcode* (line 2), a routine for constructing a NOP sled (lines 3–10), and a heap spraying attack (lines 12-18). The shellcode is a binary payload injected and executed on a user device. The NOP sled positions the shellcode in a protected memory region so that it can overwrite another running process, and execute on the target system. The final location of the shellcode is determined by a *heap spray*, which positions the NOP sled/shellcode in a specific heap location. In JavaScript this can be achieved by populating an array with fixed length copies of the NOP sled and shellcode. The heap spray is necessary to overcome address space layout randomization (ASLR) [48] protection mechanisms provided in many modern operating systems. In the final step, the attack triggers a vulnerability in the browser, or a browser plug-in causing random execution of the shellcode. A drive-by download attack is usually preceded by pre-attack JavaScript routines, which are used to deliver a suitable attack, or to conceal the attack. These pre-attack JavaScript routines include, but are not limited to: browser sniffing to identify a vulnerability in the browser

or a browser plugin; dynamic redirections to guide execution to an exploit; deobfuscation and DOM manipulation to dynamically embed scripts and vulnerable objects; or invisible iFrames to extract a hidden attack.

2.2. Machine learning algorithms

In this section, we review common supervised machine learning algorithms amenable to classification of drive-by download attacks.

Naive Bayes classifier: This simple probabilistic classifier naively applies Bayes' Theorem using a strong independence assumption between features, and predicts the class that maximizes the likelihood, which is estimated based on a given training dataset [2]. The simplicity of the algorithm makes it a suitable candidate for classification tasks with large features spaces.

Support Vector Machines (SVM): Given a training dataset consisting of benign and malicious examples, SVM produces a maximum-margin hyperplane that separates instances of two classes, maximizing the distance from any data point. The construction of the maximum-margin hyperplane requires the supporting hyperplanes to lie on a (usually sparse) subset of malicious and benign training examples, known as support vectors. The maximum-margin hyperplane is robust against slight variations in feature vectors [16], and protects against overfitting [30]. Further, the number of support vectors is independent of the number of the features, and makes SVM more robust against large feature spaces [1]. This flexible generalization capability has been validated in various theoretical and empirical studies, and makes SVM a popular candidate for classification over large feature spaces [43]. In fact, SVM is often the preferred choice for high dimensional classification problems, such as text classification, where the number of possible features can exceed the number of distinct instances available in the training data samples [8, 20].

Decision tree classifier: Decision trees transform the classification problem into a series of choices of each feature in the feature space. The most informative feature selected based on a certain criteria is used for the root node, and the dataset is divided based on the categorization at the root node. The process is recursively applied to the child nodes for subsequent subset categorizations. Pruning the subtrees based on evaluation of a test dataset is used to avoid overfitting of the classifier to the training dataset [39]. Traversing to the leaves of the decision tree provides a definitive classification for each test instance.

2.3. Evaluation Metrics

Evaluation of the effectiveness of data mining algorithms is done using standard confusion matrix metrics. The true positive rate or recall (equation 1) measures the proportion of positive cases that are correctly detected, where as the false positive rate (equation 2) specifies the proportion of negative samples that are incorrectly classified as positive. Precision is the fraction of correctly identified positive samples out of the total positive predictions (equation 3). The harmonic mean of precision and recall is given by F-measure (equation 4).

$$True\ positive\ rate\ (Recall) = \frac{True\ positives}{True\ positives + False\ negatives} \tag{1}$$

$$False\ positive\ rate = \frac{False\ positives}{False\ positives + True\ negatives} \tag{2}$$

$$Precision = \frac{True\ positives}{True\ positives + False\ positives} \tag{3}$$

$$F\text{-}measure = 2 \times \frac{precision \times recall}{precision + recall} \tag{4}$$

3. Related work

Since Provos et al. [37, 38] presented studies on the widespread abuse of drive-by downloads in web browsers, a series of research studies have been undertaken to mitigate this problem. The two general approaches employed to solve the problem are URL-based or webpage-based content analysis. Webpage content analysis can be static, dynamic (emulation based), or a combination of both. These methods can be incorporated into an off-line service in a client-server environment, or on the client-side as a semi-realtime solution embedded in a web proxy, or as a realtime solution embedded in the web browser.

3.1. Offline solutions

Off-line solutions include blacklists of malicious URLs or signature databases [52], which can be used to protect client devices. These solutions can use a variety of tools to analyze web page content, and are not restricted by execution time or computing resources. Webpage content analysis is one strategy employed by off-line solutions to detect malicious webpages. *High interaction client honeypots* monitor the entire operating system, including a web browser, using a virtual machine. When a potentially malicious URL is loaded in the sandboxed web browser, unexpected changes to the virtual machine environment can be flagged as malicious. These solutions are effective at identifying new attacks (*zero day exploits*) and have negligible false positive rates [44]. However, high interaction client honeypots can only detect malware that targets the part of client system implemented in the honeypot. Furthermore, the detection mechanism is vulnerable to various evasion techniques employed by adversaries, such as time bombs (delayed exploits triggered long after the initial visit to a web page) or logic bombs (attack is triggered after a user satisfies a set of conditions or actions).

A less resource intensive alternative is a *low interaction client honeypot*, which simulates a network protocol stack or a subset of targeted browser features in order to identify specific attack vectors [35, 45]. These solutions examine the response from a server against pre-defined rules and heuristics, or an anomaly detection classifier, in order to assess the maliciousness of a web page. Low interaction client honeypots must be instrumented to detect specific attack vectors, and therefore are not a robust solution for detecting new exploits [45].

Executing code in a controlled environment and monitoring the resulting activity was proposed as lightweight alternative to client honeypots [4]. This solution uses a manually crafted set of domain-specific features, such as website behaviour for different browser personalities, number of redirections, number of components loaded, or frequent use of string operations. Each feature is then evaluated against a pre-defined prediction model that assigns a score to each feature. If the weighted sum of the scores for a page exceeds a predefined threshold, then the page is marked malicious.

Analyzing the properties associated with webpage URLs is another strategy used in off-line solutions. A complementary approach, which derives a combination of automatically extracted, and manually created domain specific features from URL and host-based properties for supervised machine learning was proposed by Ma et al. [30].

Discovering entire malware neighbourhoods or malware distribution networks is also a viable approach in off-line detection. By identifying not only the malicious webpages, but also the landing pages linking the malicious sites, more malicious pages can be identified and blacklisted. In this approach, the data is first collected by a search engine crawler. The solution then produces a web graph of hyperlinks. While traversing the graph, the algorithm uses telemetry reports produced by anti-malware products running on client devices to discover potential malware networks [51].

Another solution uses secondary URLs, redirect chains, and malware information logged by a cluster of high interaction client honeypots to identify networks where the members are involved in distributing malware executables (malware distribution networks). For each malware distribution network, the method then discovers the servers shared by the drive-by download attacks (central servers). The URLs of these central servers are then converted into regular expression signatures. These signatures are then used by the web crawler or the browser to blacklist sites launching drive-by download attacks [54].

Efficiency is a common problem to overcome in off-line techniques. Seifert et al. [46] present an algorithmic approach to improve efficiency of high interaction client honeypots by batch processing multiple webpages simultaneously. If one or more malicious pages is found in a batch, a divide and conquer method is used to identify the suspected URLs. An alternative technique is to use static analysis to find benign webpages and eliminate them from the search space first, thus reducing the number of webpages that must be processed using more resource intensive algorithms [3, 47]. These approaches use domain specific characteristics of the exploit, the delivery mechanism of the malware to the browser, and stealth detection to identify potentially malicious webpages for further processing.

Most web-based malware can only be triggered by a specific client configuration. For example, the exploit may only target a single unpatched version of Internet Explorer. Therefore, many of the drive-by download detection mechanisms fail to identify a large proportion of malicious URLs. Kolbitsch et al. [26] proposed a technique that systematically explores all execution paths of JavaScript code as it executes. This approach has proven to be less dependent on specific client configurations, and effective in off-line environments.

Despite many advances, existing off-line solutions are resource intensive and/or not transparent. Furthermore, they are vulnerable to transience and cloaking techniques [5]. The approaches may also require additional hardware

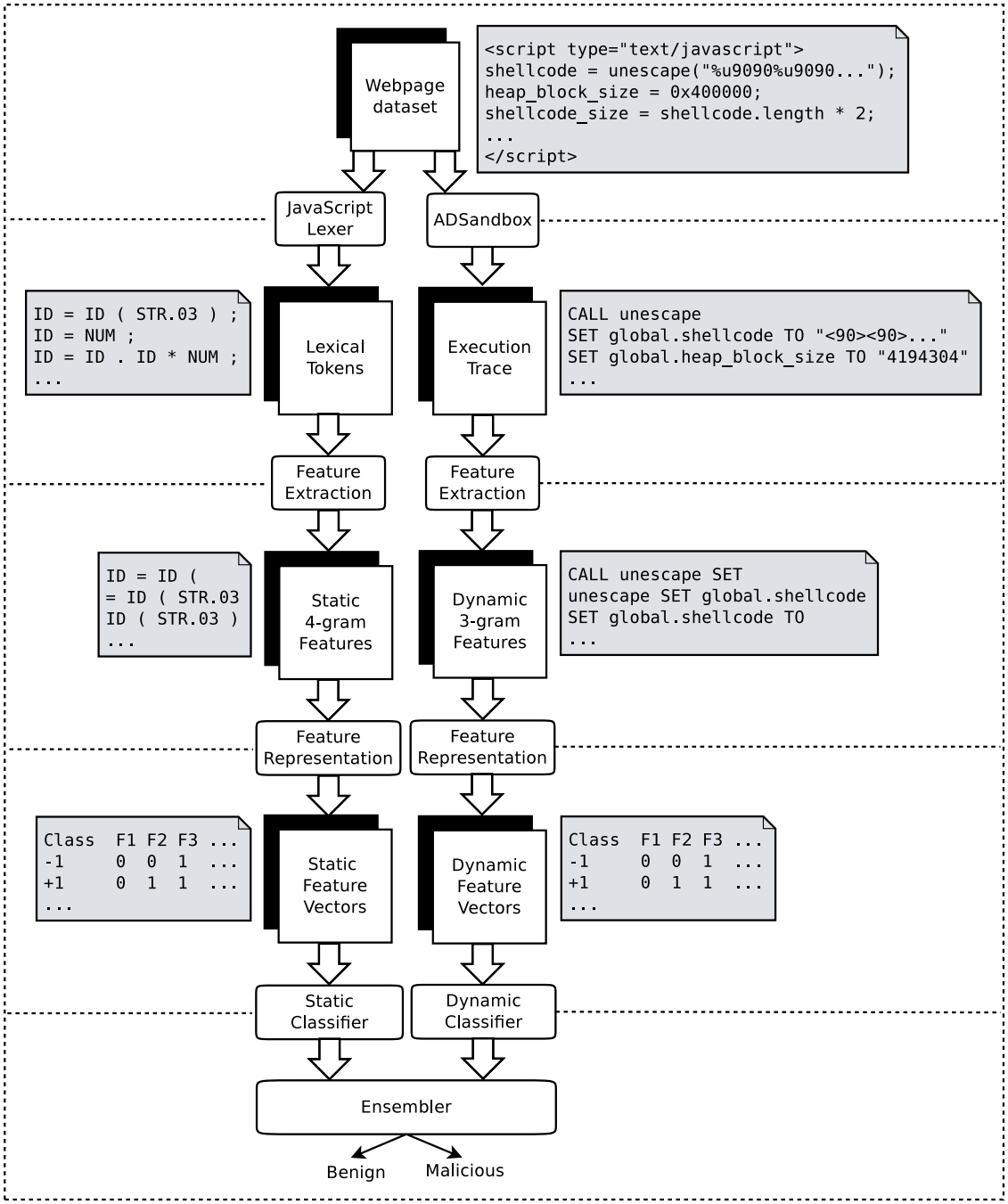

Figure 2: CUJO using ensemble classification to detect malicious webpages. CUJO decomposes embedded JavaScript in a webpage using a lexical parse of the script, as well as generating an execution trace during rendering. Both components are then processed separately, and a final, bagged classification is made using the two models.

```
CALL unescape
SET global.shellcode TO "<90><90>..."
SET global.heap_block_size TO "4194304"
SET global.shellcode_size TO "138"
SET global.spray_slide_size TO "4194110"
CALL unescape
SET global.spray_slide TO "^E^E"
SET global.spray_slide TO "^E^E^E^E"
SET global.spray_slide TO "^E^E^E^E^E^E^E^E"
...
SET global.spray_slide TO "^E^E^E^E^E^E^E^E..."
...
```

Figure 3: CUJO dynamic analysis trace.

to be maintained, and can introduce long latency periods into a normal browser session, frustrating users. As a result, these solutions are not suitable for realtime detection of drive-by download attacks on client devices.

3.2. Semi-realtime Solutions

Recent advancements in hardware have enabled the use of a subset of off-line solutions directly on the client-side. These methods run a server infrastructure, often embedded in a web proxy, and prevent users from loading malicious webpages during a browser session. One such method uses a high interaction client honeypot as described in Section 3.1 running on a proxy server to examine webpages before loading in the client web browser [32]. However, the tool may not make accurate predictions when the execution path varies due to non-deterministic factors such as randomness, time, and user input, or when the client and the honeypot environments differ [32]. For example, a page with rotating advertisements might display a benign advertisement initially, but a malicious one sometime later on in client browser. Li et al. [28] extended the work of Moshchuk et al. [32] to overcome this limitation by using a shadow browser running in a sandboxed environment in the proxy server. The client browser sends subsequent events back to the shadow browser for event handling and further processing before each load. Using sandboxes for security is a general trend being incorporated directly into the operating system by system developers [1].

A proxy resident technique that combines both static and dynamic analysis of the embedded JavaScript code to detect drive-by download attacks was proposed by Rieck et al. [42]. As illustrated in the Figure 2, the tool, called CUJO, extracts lexical tokens from the embedded JavaScript code for static analysis. For dynamic analysis, Mozilla SpiderMonkey JavaScript engine is used with a virtual browser environment to produce a log of JavaScript function calls, and state changes of JavaScript objects. Figure 3 depicts the dynamic trace produced by CUJO for the JavaScript code snippet shown in Figure 1 (a) (lines 1–10). The n-gram output from static and dynamic analysis is represented in two separate vector spaces and classified independently using a Support Vector Machine classifier. If either of the classifiers return a positive result, the web page is classified as malicious.

3.3. Realtime Solutions

Realtime approaches analyze each page on demand. Realtime solutions must be efficient and transparent to the user. These methods also tend to have strict resource restrictions. Since JavaScript is an interpreted language, static and dynamic analysis can be performed in realtime. These solutions are effective against transient and cloaking techniques that are often employed by adversaries, as specific JavaScript code paths are executed and analyzed. More fine-grained control for dynamic in-browser methods is possible by executing JavaScript in a sandbox. This can

[1] `http://www.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design`

be done with the help of readily available JavaScript interpreters, allowing malicious pages to be blocked before rendering. Dewald et al. [7] proposed one such tool, called ADSandbox, that uses a modified Mozilla SpiderMonkey JavaScript engine to log all accesses to JavaScript objects, data type conversions, and function calls that occur during interpretation of embedded JavaScript code. The traces are then matched against previously constructed regular expressions of malicious behaviour.

Curtsinger et al. [5] presented a similar approach in the ZOZZLE tool. ZOZZLE performs static analysis of deobfuscated JavaScript code in the browser. Each code segment sent to the JavaScript engine for compilation is transformed into a JavaScript abstract syntax tree (AST). Features for classification are extracted from AST nodes corresponding to expressions and variable declarations. Each feature consists of the context in which the AST node appears and the string of the AST node. As a result, ZOZZLE features capture the hierarchical structure of the code, as opposed to the flat feature space of prior methods. To select features that indicate malware presence, a χ^2 test (for one degree of freedom) with a 99.9% confidence interval is used. Finally, a Bayesian classification over the filtered hierarchical features in the AST is performed to predict maliciousness.

Lu et al. [29] take a different approach to dynamic detection of drive-by downloads. Since this class of attacks must inject a payload, Lu et al. focus analysis on only the downloaded content. Browsers use MIME types to receive data content divided into two classes: supported and unsupported. File types such as JPEG, PDF, and HTML are supported and executed transparently, while other file types such as EXE or ZIP require *user consent* before download or execution. `Blade` creates a non-executable sandbox for all supported data types, thereby preventing any binary file to execute without explicit user intervention.

Other methods target specific types of drive-by download attacks in order to balance efficiency and effectiveness. For example, approaches to detect heap spraying attacks in the browser runtime were proposed by Egele et al. [10] and Ratanaworabhan et al. [40]. Both methods detect a heap spraying attack by identifying shell code in the memory heap. Egele et al. emulate x86 assembly instructions to evaluate heap buffers (all string allocations) directly in the browser. Ratanaworabhan et al. identify possible shell codes using NOP sled detection using a tool called NOZZLE. NOZZLE samples objects allocated on the heap, and analyzes the contents concurrently to detect the NOP sled event leading up to the final shell code execution. By detecting the NOP sled, NOZZLE can abort the process before shell code execution.

Realtime detection methods must be resource conscious. Static or semi-dynamic analysis is one way to increase efficiency, but often at the cost of effectiveness. Other methods target specific attacks to improve response time. While effective for a subset of attack vectors, these approaches leave the system vulnerable to other classes of attack. Fully dynamic anomaly detection has not yet achieved widespread acceptance as a realtime solution on client devices as existing dynamic analysis techniques are only targeted at a small subset of attack vectors, or are too resource intensive.

4. Our Approach

Similar to other successful dynamic approaches to drive-by download detection, we focus on the execution of JavaScript to identify malicious activity. Our approach dynamically evaluates browser Opcode call sequences and uses supervised machine learning for classification. Figure 4 illustrates the workflow for the proposed approach. There are five distinct phases in the overall process. As shown in the diagram, an Opcode sequence is extracted from the JavaScript engine, which generates Opcodes as a part of the rendering process for each webpage. The Opcodes are then converted to a format suitable for machine learning using data reduction, feature extraction, and feature representation. In the next phase, a classifier (machine learning hypothesis or model) is trained using a seeded, labeled training set. Finally, new webpages are classified using the pre-trained learning model.

Data extraction: For comprehensive dynamic analysis, an algorithm need to capture an intermediate representation of the webpage rendering process. JavaScript Opcode (`bytecode`) is one such intermediate produced by default by the JavaScript interpreter. Since the information is generated by default, and requires no additional emulation, we log all JavaScript Opcode calls during the rendering process using a modified version of the Firefox web browser [2]. Figure 5 illustrates the disassembled browser Opcode corresponding to the malicious JavaScript code segment shown in Figure 1 (a) (lines 1–10).

[2] http://www.mozilla.com/en-US/firefox

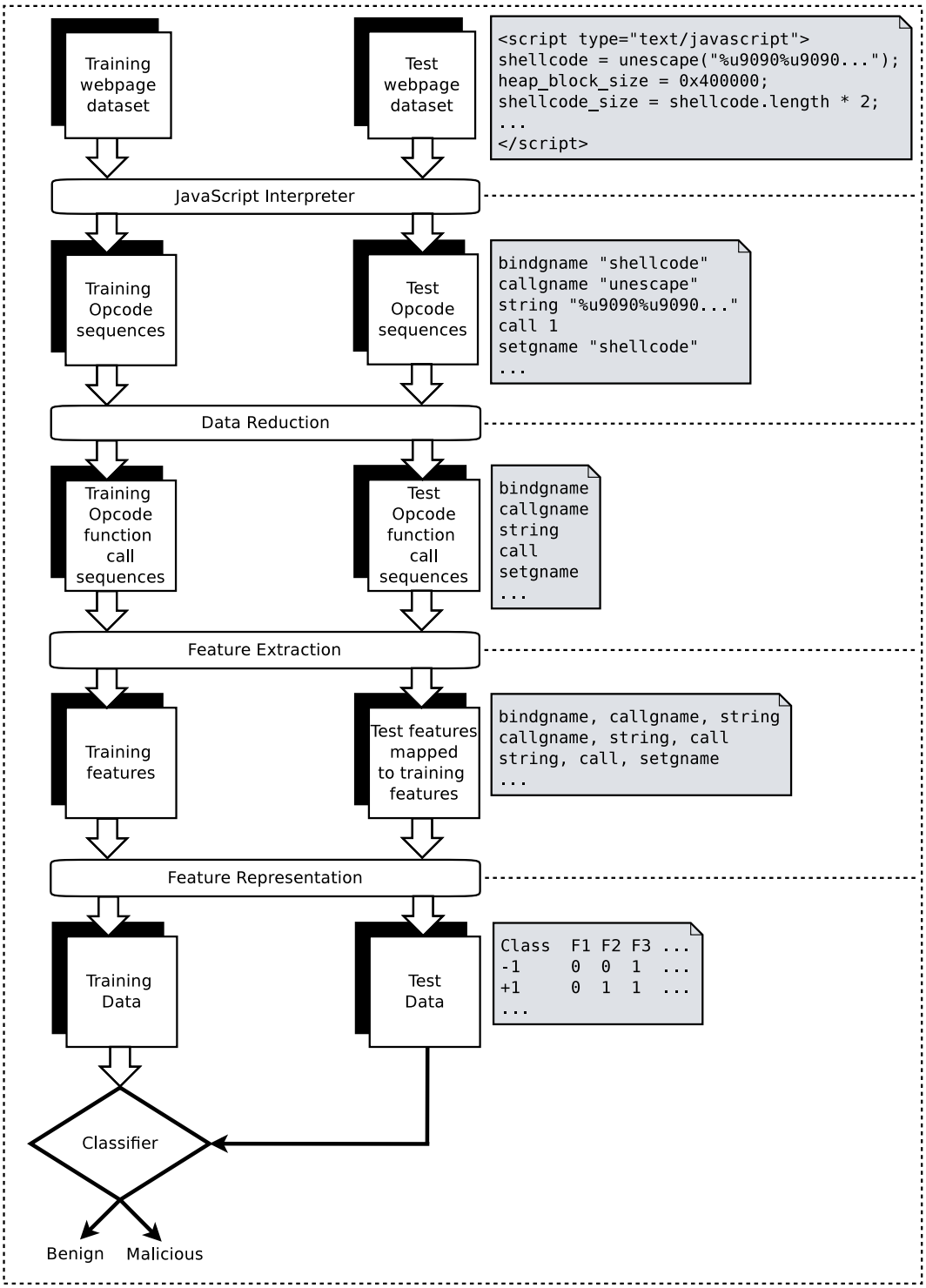

Figure 4: Opcode data processing, transformation, reduction, and classification.

```
bindgname "shellcode"
callgname "unescape"
string "%u9090%u9090..."
call 1
setgname "shellcode"
bindgname "heap_block_size"
uint24 4194304
setgname "heap_block_size"
bindgname "shellcode_size"
getgname "shellcode"
length
int8 2
mul
setgname "shellcode_size"
bindgname "spray_slide_size"
getgname "heap_block_size"
getglobal "shellcode_size"
int8 56
add
sub
setgname "spray_slide_size"
bindgname "spray_slide"
callgname "unescape"
string "%u0505%u0505"
call 1
setgname "spray_slide"
goto 94 (20)
getgname "spray_slide"
length
int8 2
mul
getgname "spray_slide_size"
lt
trace 0
bindgname "spray_slide"
getgname "spray_slide"
getgname "spray_slide"
add
setgname "spray_slide"
getgname "spray_slide"
length
int8 2
mul
...
...
getgname "spray_slide_size"
lt
trace 0
bindgname "spray_slide"
getgname "spray_slide"
getgname "spray_slide"
add
setgname "spray_slide"
getgname "spray_slide"
length
int8 2
mul
...
```

Figure 5: Disassembled Opcode generated using the Firefox JaegerMonkey JIT compiler for the JavaScript code segment shown in Figure 1 (a) (lines 1–10).

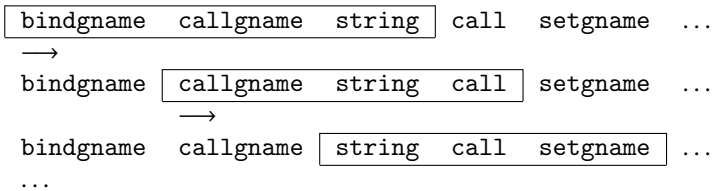

Figure 6: 3-grams extracted from the Opcode function call sequence for the trace shown in Figure 5.

Data reduction: To reduce the intermediate space usage and to improve efficiency, the function call (first most element) of each Opcode is extracted to construct a sequence of Opcode keywords. For example, the disassembled Opcode calls in Figure 5 can be represented more succinctly as "`bindgname callgname string call setgname` ...". Each distinct Opcode function call is mapped to a unique integer within the JavaScript engine. Note that each Opcode keyword is shown in human readable format in Figure 5 for clarity. The Opcode function calls are captured as a sequence of integers and extracted with no additional processing, thereby reducing the log trace overhead. Evaluating

```
ALGORITHM: ExtractFeatures(trace, map)
// Input: trace ← function call trace
// Input: map ← a map of n-grams to feature space
φ ← new Array [0 . . . σ^n − 1]
sliding_window ← new Queue()
for function_call in trace do
   sliding_window.queue(function_call)
   if sliding_window.size() == n then
      φ[map.lookup(sliding_window)] = 1
      sliding_window.dequeue()
return φ

ALGORITHM: Classify(φ, map, model)
// Input: φ ← feature space for the trace
// Input: map ← a map of n-gram to feature space index
// Input: model ← ω
f_x = 0
for i ← 0 to map.keys().size() - 1 do
   f_x = f_x + model[i] ∗φ[i]
return sign(f_x)
```

Figure 7: Feature extraction and classification using Opcode traces.

only function keyword sequences has also be successfully deployed at the kernel level for anomaly detection in operating systems [12, 13, 34, 53].

Feature extraction: Since a single browser session can be composed of many successive page loads, we treat the intermediate call trace as a data stream, and extract features for machine learning using a single pass sliding window [6]. As the window progresses through the stream we extract n-grams of Opcode keywords to better capture order of execution within the training and prediction model. The use of n-grams in the sliding window model is a widely studied approach to improve overall effectiveness in machine learning and anomaly detection [27, 36, 42].

For example, a sliding window of length three over the Opcode keywords from Figure 5 produces 3-grams shown in Figure 6. Each unique n-gram results in a distinct feature in the overall feature space used for the machine learning model. Since increasing n can also dramatically increase the size of the overall feature space, the most appropriate value of n is often determined empirically in order to find an acceptable effectiveness and efficiency trade-off.

Feature representation: The n-grams for the features are represented as a vector space, where each unique n-gram constitutes a dimension. So, the maximum size of the feature space is bounded above by σ^n, where σ is the number of distinct Opcode function calls in the training dataset. Presence or absence of each n-gram in the trace is represented in a sparse vector space using binary representation. However, the unique n-grams for a given webpage can never be greater than the length of the Opcode call trace in practice. Let m represent the length of an Opcode call trace for a given webpage. While the potential feature space is very large, the total number of distinct n-grams is linear with respect to the length of the JavaScript call trace for a rendered webpage since the trace can have at most $m - n + 1$ distinct n-grams. Therefore, the non-zero n-grams in the vector space is $s \in O(m - n)$ in the worst case.

Data mining: Finally, a linear Support Vector Machine (SVM) algorithm (libLINEAR [11]) is used to derive an appropriate hyperplane, which can then be used to make predictions on future data samples. Let $x_i \in R^{\sigma^n}$, $i = 1, ..., \ell$ be the feature vectors, and a vector $y \in R^\ell$ such that $y_i = \{-1, 1\}$ represent the labels for benign and malicious classes respectively for the ℓ training examples. Given x_i and y_i, $i = 1, ..., \ell$, libLINEAR constructs a weight vector ω representing the maximum-margin hyperplane, such that $f(x) = \omega^T.x$, where x is the vector space of n-grams for the sample to be classified and $sign(f(x))$ is the decision function. The Figure 7 presents the pseudocode for Opcode analysis feature extraction and classification algorithms.

The input to the data mining algorithm captures the execution sequence of the JavaScript Opcode function calls. The features for machine learning are derived from the function calls along with the context they occur using the n-gram model. As the features are derived from a fixed set of tokens, the overall feature coverage is high, and the probability of unseen features in the test classifier is low. Therefore, the machine learning algorithm has sufficient data to make accurate predictions on previously unseen traces since most of the feature space is represented in the training model.

Worst Case Analysis: The total length of an Opcode call trace (m) for a given webpage is linear with respect to the number of instructions executed by the embedded JavaScript code. As such, the worst case efficiency of the Opcode analysis technique is dependent on the inherent efficiency of the JavaScript code. If the JavaScript code is efficient, the generated Opcode trace is very small, and subsequently the number of n-grams extracted is small. However, if for example the code being executed has a length of N, and has a worst case efficiency of $O(N!)$, the number of Opcode tokens generated can also be $O(N!)$ which is potentially catastrophic. While this seems to be significant limitation, this problem generally applies to any dynamic method that requires JavaScript code to be executed before processing.

Thankfully, just-in-time JavaScript compilers now have several optimizations such as maximum recursion depth, and timed execution truncation to prevent unreasonable response times of embedded code. See, for example Gal [14], Gal et al. [15], Sol et al. [49] and the references therein for further details. The Opcode dynamic analysis implicitly benefits from all of these runtime optimizations. Other pragmatic enhancements to the dynamic approach such as automatically enforcing a maximum length for any given Opcode call trace is also possible, but not explored further in this work.

As shown in Figure 7, feature extraction uses a hash table containing at most $S \in O\left(\sum_M (m - n + 1)\right)$ elements, where M is the number of training examples, and $\sum_M (m - n + 1) < \sigma^n$. Given a training dataset comprised of s non-zero features per example, a linear SVM can be trained in $O(sM)$ time [21]. The linear SVM classifier performs at most S additions, and multiplications for each classification. Therefore, the worst case runtime and space efficiency for classification using linear SVM is $O(S) + O(m)$ for each Opcode call trace evaluated.

5. Evaluation

In this section, the effectiveness and efficiency of the proposed method is evaluated, and compared with existing state of the art drive-by download detection techniques. All experiments are conducted on a computer with an Intel® Core™ 2 Duo 3.16GHz processor and 4 GB of RAM, running the Ubuntu 10.10 operating system.

5.1. Datasets

To the best of our knowledge, no publicly available gold standard test collection for JavaScript malware detection exists. Therefore, a benign web page dataset, and a malicious web page dataset were constructed. The benign dataset contains landing pages for 10, 620 sites listed as most popular by Alexa[3], collected in June 2011. Therefore, the dataset is representative of many present use of JavaScript. Each page of the benign dataset was validated with Google safe browsing API and standard antivirus solutions. The malicious web page dataset was created by downloading URLs and using samples published in different malware forums [4,5,6,7]. The downloaded webpages from these URLs may not contain drive-by download attacks, due to transient and cloaking techniques employed by malicious sites. Therefore, Google safe browsing API and antivirus solutions were used to flag pages believed to be malicious. To avoid false positives, only the pages identified as malicious by at least 5 screening sources were marked as malicious. A total of 57 distinct attack vectors were identified in the collection.

Note that the use of antivirus solutions may remove the latest attacks from the dataset. But, this is not problematic as the objective is not to find latest attacks, but to evaluate classification algorithms that do not contain domain specific intervention. Each data point is verified to contain a unique, complete exploit, or pre-attack JavaScript routines using

[3] http://www.alexa.com/topsites
[4] http://www.malwaredomainlist.com/
[5] http://www.malwaredomains.com/
[6] http://sucuri.net/
[7] http://www.blade-defender.org/

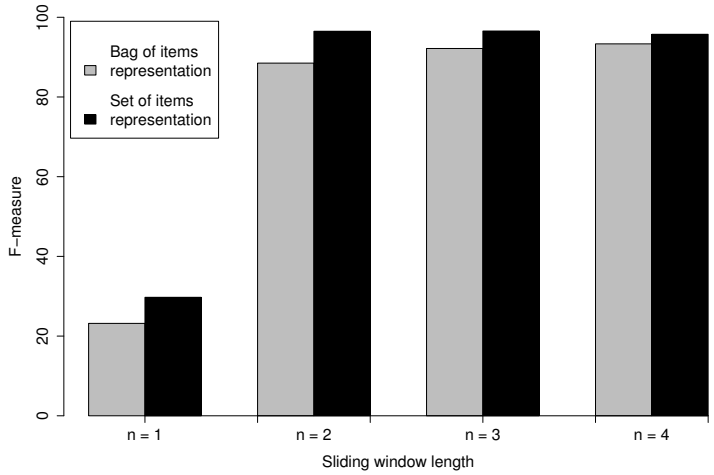

Figure 8: Classification effectiveness (F-measure) for "bag of items" and "set of items" feature representations using a SVM classifier.

Algorithm	n	TP Rate (Recall) (%)	FP Rate (%)	Precision (%)	F-Measure (%)
Naive Bayes	4	100.0	0.67	44.53	61.62
J48	2	96.49	0.03	94.83	95.65
SVM	3	98.25	0.03	94.92	96.55

Table 1: Sliding window length (n) and algorithmic effectiveness of Opcode analysis using alternative classification algorithms.

attack categories given by the antivirus solutions and manual examination of the deobfuscated JavaScript code. If more than one page contained the same attack vector, it was removed from the collection. The goal of the pruning is to ensure that the classification make accurate predictions on unseen exploits. If only one representative of each exploit exists, it cannot appear in both the training and test partitioning, and we can be confident that the predictive power of the algorithm is not biased. The data points containing pre-attack JavaScript routines are included in order to identify an attack as early as possible, making intervention at runtime more tractable.

5.2. Experimental methodology

For evaluation, stratified 10−fold cross validation is used. The dataset is randomly divided into 10 equal partitions, maintaining the same proportion of benign to malicious samples in each partition. Then, ten runs of 1 test : 9 training classifications are ran, and the results averaged.

5.3. Parameter selection for Opcode *analysis*

Opcode analysis depends on certain design parameters for good effectiveness and efficiency. Some of these design parameters require empirical analysis and are outlined here:

- Small values of n can result in high bias due lack of predictive signals, while large values of n can increase variance due to overfitting. Increasing n creates an exponentially larger feature space. Therefore, the smallest sliding window length with the desired accuracy can be determined empirically on the training set. In prior work, feature values were represented as presence or absence of a feature in the trace (*set of items representation*) [42], or as a frequency weighting scheme, where the number of appearances of a feature in the trace is used (*bag of items representation*) [19, 24]. The bag of items representation can lead to increased variance, while the set of items representation can result in high bias. Figure 8 shows the classification effectiveness in terms of F-measure for both representations with a SVM classifier on the dataset presented in the Section 5.1.

Algorithm	n	TP Rate (Recall) (%)	FP Rate (%)	Precision (%)	F-Measure (%)
CUJO static	4	82.46	0.02	95.92	88.68
CUJO dynamic	3	77.19	0	100.0	87.13

Table 2: Sliding window length (n) and algorithmic effectiveness with stratified 10-fold cross validation for the CUJO algorithm on the combined benign dataset described in Section 5.1 and the malicious dataset comprised of more than one sample of the same antivirus category.

The results indicate that the set of items representation with a sliding window of length 3 produces good overall effectiveness.

- Different data mining algorithms can be utilized for classification. Table 1 presents the algorithmic effectiveness for Opcode analysis with a Naive Bayes classifier, a J48 decision tree classifier and a linear SVM classifier using the best values of n (determined experimentally). As expected, SVM consistently produces the best overall effectiveness on the dataset.

5.4. Baselines

As the baseline JavaScript malware detection techniques, ADSandbox [7], and CUJO [42] algorithms were re-implemented for the test environment. Furthermore, PhoneyC [35] was installed using the latest code available at the time of the experiments. Since several of the baselines were implemented independently, we outline the details of our baselines.

- ADSandbox employs a misuse detection approach with regular expression signatures to find malicious webpages. Since we do not have access to the original signatures, we parse a sliding window of length n over the training data partition, and use n-grams unique to malicious webpages as signatures to classify items in the test data partition.
- In addition to the JavaScript interpreter, CUJO and ADSandbox require a virtual browser environment to properly execute the embedded JavaScript. The completeness of virtual browser environment is a trade-off between performance and effectiveness. In the re-implementation of the algorithms, a near complete virtual browser environment is used to reduce JavaScript errors that might otherwise arise.
- Similar to Opcode analysis, the sliding window lengths for CUJO static, CUJO dynamic, and ADSandbox algorithms were determined empirically in order to maximize the effectiveness.
- For time and space evaluation of CUJO and the Opcode algorithms, the implementations used a similar processing pipeline, as sketched in Figure 7.

ADSandbox in our environment relies on dynamically generated signatures from the training dataset, while PhoneyC uses a set of predefined static heuristics to detect malicious shellcode and heap spray attacks. As such, PhoneyC did not detect malicious activity in many of the test samples, especially if the sample contained only pre-attack JavaScript routines. The low detection rate of ADSandbox and PhoneyC highlight the difficulty of requiring manual generation of attack vector signatures, and the inability of these approaches to make accurate predictions on previously unseen attack types.

CUJO does not rely on signatures, but did not perform as well as anticipated in our test environment. A high statistical variance in accuracy between different datasets, and a lower accuracy for dynamic analysis than static analysis on most of the datasets was originally observed in the experiments presented by Rieck et al. [42]. Since the original experiments of Rieck et al. [42] included multiple instances of each attack category, we reconstructed our malicious instances to include multiple malicious sample pages from each attack category to ensure the consistency of our baseline implementation.

Table 2 shows the effectiveness of CUJO on the combined benign dataset described in Section 5.1 and a dataset containing multiple malicious instances of each attack type. On this dataset the CUJO static and dynamic algorithms show a dramatic increase in true positive rate and a reduction in the false positive rate, and provide similar results to the original experiments performed by Rieck et al. [42].

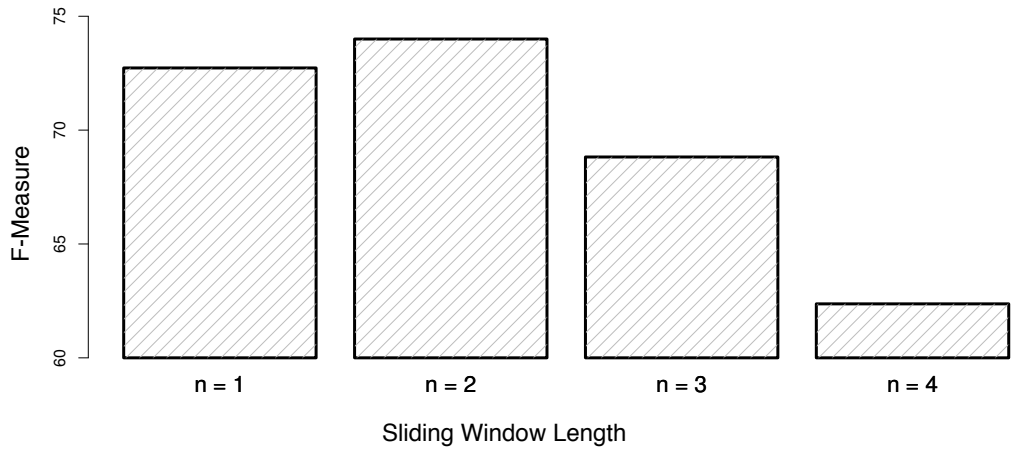

Figure 9: Classification effectiveness (F-measure) for CUJO dynamic algorithm using a SVM classifier for varying sliding window lengths (n).

In Figure 9 is the effectiveness of CUJO dynamic algorithm on the dataset given in Section 5.1. The CUJO dynamic achieved the best effectiveness when the sliding window length is 2, and the detection rate decreased with increasing n. Having the best detection rate at a smaller sliding window length, compared to the original experiment [42], and the deterioration of the effectiveness along with increasing sliding window lengths, can be indicative of the CUJO dynamic algorithm's effort to reduce variance on a dataset with unique malicious samples.

An examination of a few malicious examples that were classified by the CUJO dynamic algorithm as benign revealed that the main reason for erroneous classification of those malicious items is the sample specific values (JavaScript variable names, and values) are present in the malicious samples, but not in the training set. As a result, many of the 3-grams are unique, and therefore only appear in the training or the test partition, but not both. For example, consider the CUJO dynamic trace shown in the Figure 3. If the user defined JavaScript variable names and values are not seen in any of the examples in the training dataset, and if a sliding window of length 3 is selected, only "`CALL`, `unescape`, `SET`" 3-gram from the attack code segment would take part in the classification process. In fact, out of all 3-gram features in malicious instances, 40% contained a user defined variable name and 98% contained a user defined variable name or value.

5.5. Feature space

The growth of the feature space along with the increasing dataset sizes for the Opcode analysis and the CUJO dynamic algorithm is shown in Figure 10. CUJO dynamic has a much larger feature space that grows along with the size of the dataset, which is caused by the inclusion of sample specific values. Figure 11 shows the number of data samples in which a feature is present, for features with a weight inclined towards a classification decision as malicious in the weight vector ω, ordered by the count of data samples containing the feature from most frequent to the least. CUJO dynamic demonstrates a high dimensional feature space with low coverage, which increases with n. The high dimensionality with low coverage is problematic for statistical learning methods. As such, the lower detection rate of CUJO dynamic algorithm in our current environment can be partially attributed to using a dataset with only unique malicious samples.

CUJO dynamic analysis is promising for predicting "maliciousness" of unseen attacks, but suffer from one major drawback. The elements of feature composition are not fixed. As a result, the CUJO dynamic algorithm generates a large feature space with low coverage. Variable feature composition does not pose a serious limitation if the token space adheres to Heaps law, which states that there will be diminishing returns in terms of new feature discovery along with growing training dataset sizes. But in malware detection, where adversaries actively search for ways to

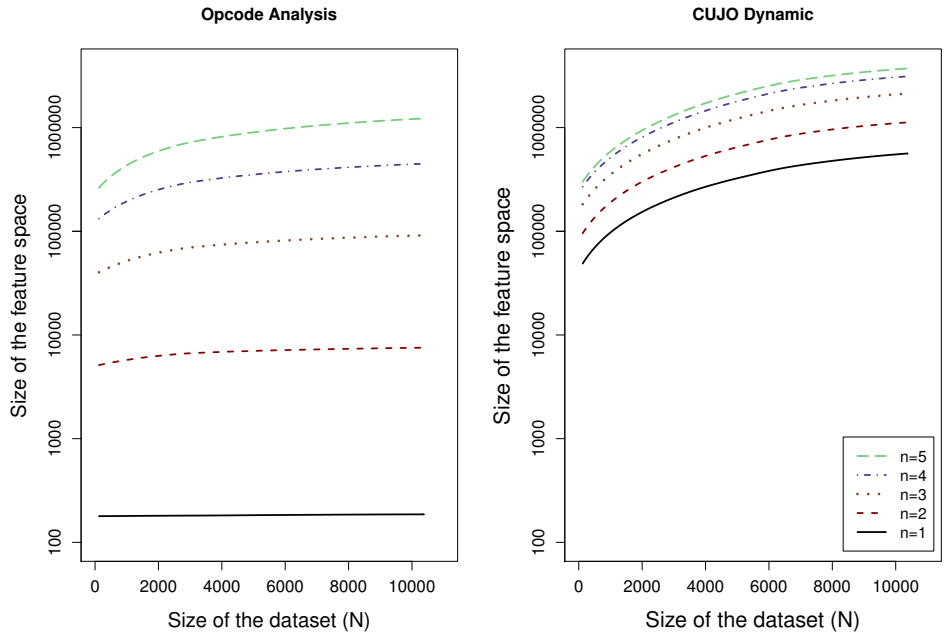

Figure 10: Growth in the feature space of Opcode and CUJO dynamic with increasing dataset size (N) for different n.

Algorithm	n	TP Rate (Recall) (%)	FP Rate (%)	Precision (%)	F-Measure (%)
Opcode analysis	3	98.25	0.03	94.92	96.55
ADSandbox	1	82.46	2.03	17.87	29.38
CUJO static	4	70.18	0.06	86.96	77.67
CUJO dynamic	2	64.91	0.06	86.05	74.00
CUJO ensembled		84.21	0.11	80.0	82.05
PhoneyC		31.58	0.00	100.0	48.00

Table 3: Effectiveness of Opcode analysis and existing state of the art detection methods with stratified 10-fold cross validation on the dataset described in Section 5.1. The values of n presented achieve the highest accuracy for any n value between 1 and 4.

circumvent detection mechanisms, inclusion of sample specific values, especially user defined variable names can cause unexpected behaviour. In fact, in the case of the CUJO dynamic analysis, a considerable portion of the feature space that contribute towards a malicious classification can be avoided just by changing the JavaScript variable names used in the attack. The CUJO static algorithm feature space has higher coverage. But the predictability of the static algorithm is often low due to code obfuscation techniques employed by many adversaries.

The feature space of the Opcode method only contains Opcode function calls, where the number possible of Opcode functions (σ) is fixed. As a result, the feature space automatically extracted from the training dataset is generic and can never grow larger than σ^n. In contrast, the CUJO dynamic algorithm allows variable names and assignments to be valid tokens in the n-gram feature space [42]. Moreover, CUJO augments the automatically generated feature space with a pre-matching filter of previously identified n-grams previously known to precede specific attacks. However, the prematching requires domain specific knowledge, and is equivalent to using signatures to identify well-known attack vectors. The Opcode method uses no domain specific pre-filtering, and automatically extracts the feature space from the training dataset, and requires no user intervention.

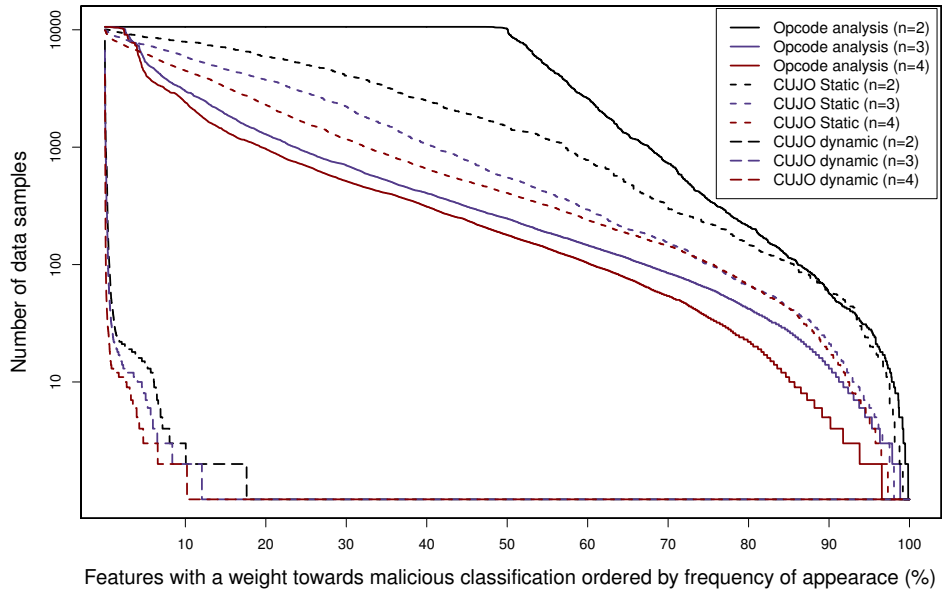

Figure 11: Number of distinct n-grams per training instance.

5.6. Effectiveness and efficiency

Table 3 presents the sliding window length (n) and the corresponding effectiveness for all four approaches on the dataset described in Section 5.1. The given sliding window lengths produce the best results under 10-fold cross validation, and hence can be considered as the maximum-likelihood estimation (MLE) of n for the optimal classifier. The results demonstrate the high effectiveness of Opcode analysis compared to the other methods on the dataset described in Section 5.1. The Opcode analysis has a high true positive rate and a very low false positive rate. For intrusion detection applications both these qualities are important, as a low true positive rate implies a high probability of a compromise, where as a high false positive rate can result in lose of user confidence due to frequent false interruptions.

Any dynamic analysis technique, irrespective of the methodology, requires the code to be emulated. Opcode analysis has been designed to perform the analysis using a by-product of the main browser process, so that when deployed as an in-browser solution can piggyback on the main browser process with negligible data extraction overhead, preventing duplication of the emulation process. The CUJO dynamic algorithm extends hook functions provided by the JavaScript interpreter to generate the dynamic analysis trace, which require the JavaScript engine to be built with additional parameters, that add delays to the running time. However, for simplicity, and to illustrate benefits and trade-offs of Opcode analysis, the CUJO static and dynamic analysis logs are presumed to be generated from the same browser emulation process. This allows the comparison of additional time and space used by the core of the anomaly detection algorithm, which is the feature extraction and classification.

Figure 12 illustrates time overhead, and shows the effectiveness versus space overhead incurred on feature extraction and classification for Opcode analysis, CUJO static and CUJO dynamic analysis algorithms on the dataset given in the Section 5.1. The Opcode analysis has a very low memory utilization compared to the CUJO algorithm, mainly because the derived features in an Opcode trace have a fixed vocabulary, but the feature space of CUJO dynamic grows with the number of training instances. As the graphs reveal, Opcode analysis trades time efficiency for better effectiveness and lower space utilization. Opcode analysis is highly effective and space efficient compared to existing dynamic anomaly detection techniques and performs the analysis within an acceptable time.

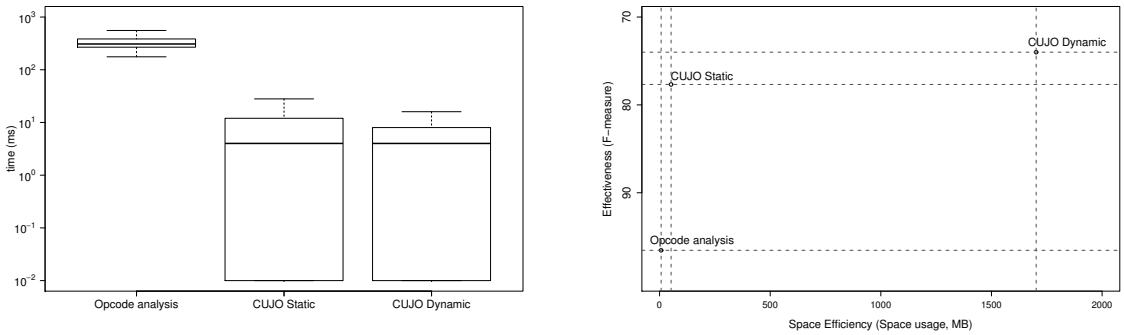

Figure 12: Feature extraction and classification time of Opcode analysis and CUJO static/dynamic algorithms (left), and effectiveness (F-measure) versus memory space usage (right) for the dataset given in Section 5.1.

Weight	Feature	% appearance in benign samples	% appearance in malicious samples
0.0294	`return, getthisprop, not`	6.1	64.2
0.0245	`setlocal, moreiter, ifne`	75.6	100.0
0.0245	`forlocal, getarg, getlocal`	80.0	100.0
0.0245	`setlocal, getarg, iter`	80.0	100.0
0.0245	`trace, forlocal, getarg`	88.4	100.0
0.0222	`string, setlocal, goto`	17.4	70.2
0.0220	`getthisprop, setelem, getlocal`	72.5	98.2
0.0220	`int8, zero, call`	72.5	98.2
0.0220	`getlocal, one, uint16`	72.4	98.2
0.0220	`one, uint16, zero`	72.4	98.2

Table 4: Top 10 weights towards malicious classification for Opcode analysis and their coverage in the dataset presented in the Section 5.1

5.7. Training data requirements

Figure 13 show the impact of increasing training instances on effectiveness for CUJO and Opcode. In Figure 13 (left), ten fold-cross validation is used, and therefore a minimum of 10 malicious instances are required – one for each fold. The Opcode method has a remarkably stable learning curve when using ten fold cross-validation, while CUJO algorithms benefit from increased training instances. Therefore, in Figure 13 (right) we evaluated a straight 1:2 test and train partitioning of malicious dataset to determine exactly how sensitive the Opcode method is to malicious training instances. The entire benign dataset is included in the training partition and malicious instances in the training partition is incremented by 2 samples at a time. This procedure is repeated with random partitioning of the malicious dataset for 5 times and the results are averaged. The Opcode algorithm is remarkably effective with very few malicious training instances.

5.8. Evasion

Table 4, shows the 10 features with the highest weight towards malicious classification in Opcode analysis. As shown in the graph and the table, the Opcode analysis uses a feature space with a high coverage and as a result large number of features with small weights take part in the classification of each sample. As such, Opcode analysis is less susceptible to such attacks. However, in system call analysis, methods to mount attacks by executing legitimate system call sequences have been shown to be effective in prior work [34]. Similar strategies could be used against Opcode

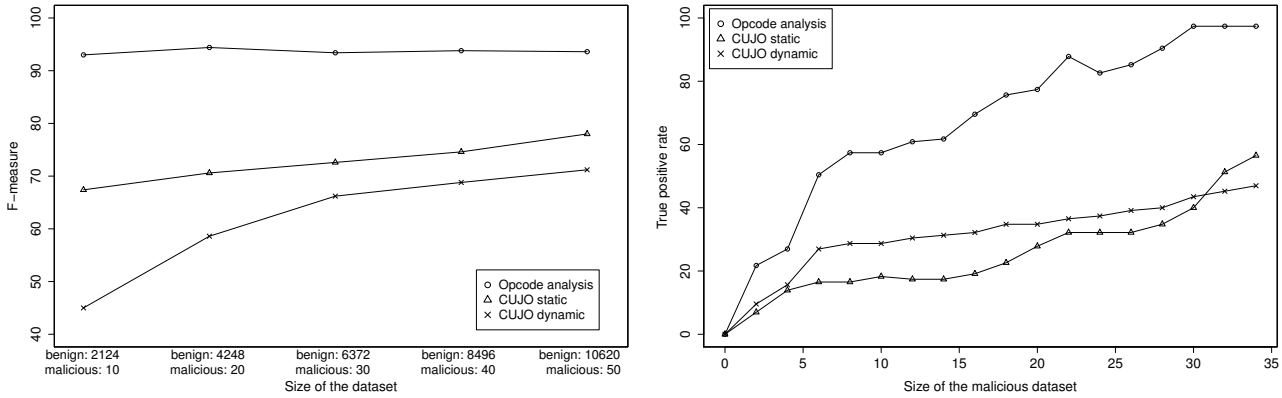

Figure 13: The effect of increasing the number of training instances for Opcode analysis, and CUJO on F-measure and the True Positive Rate. The learning curve on the left uses proportional growth of malicious to benign training instances, while the curve on the right uses the entire benign dataset and $2/3$s of the malicious samples for training, and remaining $1/3$ of the malicious samples constitute the test dataset.

analysis. Retraining the algorithm with an up-to-date dataset, and including features derived from arguments passed to Opcode function calls may be an option to reduce the likelihood that such attack would circumvent detection.

6. Conclusions and Future Work

Previous dynamic anomaly detection techniques with automated feature extraction to detect drive-by download attacks do not attempt to limit the size and coverage of the training feature space. As a result, these approaches can be less effective in anomaly detection, and resource intensive. Opcode analysis addresses this issue by using a bounded feature space with high coverage. As a result, Opcode analysis is effective and efficient, with a low space utilization, and has an acceptable runtime overhead.

As with other dynamic anomaly detection techniques, Opcode analysis is bound by the underlying efficiency of the JavaScript code being executed, and is still susceptible to carefully crafted evasion techniques. Reducing chances of potential evasion, and deploying Opcode analysis on resource constrained client devices remain future research. Reducing chances of an evasion is critical to any security application. Static (E.g. CUJO static), and semi-dynamic (E.g. ZOZZLE) approaches also face the threat of evasion [5]. The static, semi-dynamic, and dynamic techniques are complementary to each other, and differ in resource utilization as detection is performed at different stages of the JavaScript execution. Joint classification with static, semi-dynamic, and dynamic approaches is another possibility to consider to further reduce resource utilization, and chances of evasion. The viability of such approaches in resource constrained environments is an important future research direction.

Additional research is necessary before Opcode analysis on client devices can be fully realized. Similar to the approach taken by Lu et al. [29], redirecting all downloads to a non-executable sandbox, and mapping only the downloads with a non-malicious Opcode trace to the file system may be one way to use Opcode analysis on resource constrained client devices. Also, the JavaScript just-in-time execution traces are likely to differ between various browser implementations, and is another factor to be considered before attempting to deploy this technique on client devices.

In summary, the low training data requirements combined with automatic feature extraction make Opcode analysis an attractive alternative. The approach requires no background knowledge of JavaScript malware, and is simple to use. Opcode analysis provides a light-weight effective semi-realtime solution for drive-by download detection and prevention.

References

[1] Ayat, N. E., Cheriet, M., Suen, C., May 2005. Automatic model selection for the optimization of svm kernels. Pattern Recognition 38 (10), 1733–1745.

[2] Bishop, C., 2006. Pattern Recognition and Machine Learning. Springer.

[3] Canali, D., Cova, M., Vigna, G., Kruegel, C., March 2011. Prophiler: a fast filter for the large-scale detection of malicious web pages. In: Proceeding of the 20th international conference on World wide web (WWW '11). Hyderabad, India, pp. 197–206.

[4] Cova, M., Kruegel, C., Vigna, G., April 2010. Detection and analysis of drive-by-download attacks and malicious javascript code. In: Proceeding of the 19th international conference on World Wide Web (WWW '10). Raleigh, North Carolina, USA, pp. 281–290.

[5] Curtsinger, C., Livshits, B., Zorn, B., Seifert, C., August 2011. ZOZZLE: Low-overhead mostly static javascript malware detection. In: Proceeding of the 20th USENIX security symposium (USENIX '11). San Francisco, CA, USA.

[6] Datar, M., Gionis, A., Indyk, P., Motwani, R., 2002. Maintaining stream statistics over sliding windows: (extended abstract). In: Proceeding of the 13th annual symposium on Discrete algorithms (SODA '02). Philadelphia, PA, USA, pp. 635–644.

[7] Dewald, A., Holz, T., Freiling, F. C., March 2010. ADSandbox: Sandboxing javascript to fight malicious websites. In: Proceeding of the 25th ACM Symposium on Applied Computing (SAC '10). Sierre, Switzerland, pp. 1859–1864.

[8] Drucker, H., W., D., Vapnik, V., September 1999. Support vector machines for spam categorization. IEEE Transactions on Neural Networks 10 (5), 1048–1054.

[9] Egele, M., Kirda, E., Kruegel, C., 2009. Mitigating drive-by download attacks: Challenges and open problems. In: Open Research Problems in Network Security – iNetSec 2009. Vol. 309 of IFIP Advances in Information and Communication Technology. Springer Boston, pp. 52–62.

[10] Egele, M., Wurzinger, P., Kruegel, C., Kirda, E., 2009. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In: Detection of Intrusions and Malware, and Vulnerability Assessment. Vol. 5587 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 88–106.

[11] Fan, R., Chang, K., Hsieh, C., Wang, X., Lin, C., June 2008. LIBLINEAR: A library for large linear classification. Journal of Machine Learning Research 9, 1871–1874.

[12] Forrest, S., Hofmeyr, S., Somayaji, A., December 2008. The evolution of system-call monitoring. In: Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC '08). Anaheim, California, USA, pp. 418–430.

[13] Forrest, S., Hofmeyr, S. A., Somayaji, A., Longstaff, T. A., May 1996. A sense of self for Unix processes. In: Proceeding of the 17th IEEE Symposium on Security and Privacy (S&P '96). pp. 120–128.

[14] Gal, A., 2006. Efficient bytecode verification and compilation in a virtual machine. Ph.D. thesis, Irvine, CA, USA.

[15] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E. W., Reitmaier, R., Bebenita, M., Chang, M., Franz, M., 2009. Trace-based just-in-time type specialization for dynamic languages. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI '09). Vol. 44. ACM, Dublin, Ireland, pp. 465–478.

[16] Hamel, L., 2009. Knowledge Discovery With Support Vector Machines. Wiley Series on Methods and Applications in Data Mining. John Wiley & Sons.

[17] Hawkins, D. M., 2004. The problem of overfitting. Journal of Chemical Information and Modeling 44 (1), 1–12.

[18] Hoffman, B., Sullivan, B., 2008. Ajax security. Safari Books Online. Addison-Wesley.

[19] Hu, W., Liao, Y., Vemuri, V. R., 2003. Robust anomaly detection using support vector machines. In: In Proceedings of the International Conference on Machine Learning.

[20] Joachims, T., 1998. Text categorization with support vector machines: Learning with many relevant features. In: Machine Learning: ECML-98. Vol. 1398 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 137–142.

[21] Joachims, T., 2006. Training linear svms in linear time. In: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, pp. 217–226.

[22] John, P. J., Yu, F., Xie, Y., Krishnamurthy, A., Abadi, M., March 2011. Heat-seeking honeypots: Design and experience. In: Proceeding of the 20th International Conference on World Wide Web (WWW '11). Hyderabad, India, pp. 207–216.

[23] Johns, M., 2008. On javascript malware and related threats. Journal in Computer Virology 4, 161–178.

[24] Kang, D., Fuller, D., Honavar, V., June 2005. Learning classifiers for misuse and anomaly detection using a bag of system calls representation. In: Proceeding of the 6th Annual IEEE SMC Information Assurance Workshop (IAW '05). pp. 118–125.

[25] Kaplan, S., Livshits, B., Zorn, B., Seifert, C., Curtsinger, C., 2011. "NOFUS: Automatically Detecting" + String.fromCharCode(32) + "ObFuSCateD ".toLowerCase() + "JavaScript Code". Tech. rep., Microsoft.

[26] Kolbitsch, C., Livshits, B., Zorn, B., Seifert, C., October 2011. Rozzle: De-cloaking internet malware. Tech. rep., Microsoft.

[27] Lee, W., Stolfo, S. J., January 1998. Data mining approaches for intrusion detection. In: Proceedings of the 7th Conference on USENIX Security Symposium (USENIX '98). Vol. 7. San Antonio, Texas, USA, pp. 6–20.

[28] Li, Z., Tang, Y., Cao, Y., Rastogi, V., Chen, Y., Liu, B., Sbisa, C., February 2011. WebShield: Enabling various web defense techniques without client side modifications. In: Proceeding of the 18th Annual Network & Distributed System Security Symposium (NDSS '11). San Diego, California, USA.

[29] Lu, L., Yegneswaran, V., Porras, P., Lee, W., October 2010. BLADE: an attack-agnostic approach for preventing drive-by malware infections. In: Proceedings of the 17th ACM conference on Computer and communications security (CCS '10). Chicago, Illinois, USA, pp. 440–450.

[30] Ma, J., Saul, L. K., Savage, S., Voelker, G. M., 2009. Beyond blacklists: learning to detect malicious web sites from suspicious urls. In: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '09). Paris, France, pp. 1245–1254.

[31] Marsland, S., 2009. Machine Learning: An Algorithmic Perspective. Chapman & Hall/CRC machine learning & pattern recognition series. CRC Press.

[32] Moshchuk, A., Bragin, T., Deville, D., Gribble, S. D., Levy, H. M., August 2007. SpyProxy: execution-based detection of malicious web content. In: Proceeding of the 16th USENIX Security Symposium (USENIX '07). Boston, MA, USA.

[33] Moshchuk, A., Bragin, T., Gribble, S. D., M., L. H., February 2006. A crawler-based study of spyware on the web. In: Proceeding of the 13th Annual Symposium on Network and Distributed System Security (NDSS'06). San Diego, California, USA.
[34] Mutz, D., Valeur, F., Vigna, G., Kruegel, C., February 2006. Anomalous system call detection. ACM Transactions on Information and System Security (TISSEC) 9, 61–93.
[35] Nazario, J., 2009. Phoneyc: a virtual client honeypot. In: Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more (LEET' 09). Boston, MA, USA, pp. 6–13.
[36] Perdisci, R., Gu, G., Lee, W., December 2006. Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems. In: Proceeding of the 6th International Conference on Data Mining (ICDM '06). Hong Kong, pp. 488–498.
[37] Provos, N., Mavrommatis, P., Rajab, M. A., Monrose, F., July 2008. All your iframes point to us. In: Proceeding of the 17th conference on USENIX Security Symposium (USENIX SS'08). San Jose, CA, USA, pp. 1–15.
[38] Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N., April 2007. The ghost in the browser analysis of web-based malware. In: Proceedings of the 1st Workshop on Hot Topics in Understanding Botnets (HotBots '07). Cambridge, MA.
[39] Quinlan, J., 1993. C4.5: Programs for Machine Learning. Morgan Kaufmann Series in Machine Learning. Morgan Kaufmann Publishers.
[40] Ratanaworabhan, P., Livshits, B., Zorn, B., April 2009. NOZZLE: a defense against heap-spraying code injection attacks. In: Proceedings of the 18th conference on USENIX security symposium (SSYM '09). Montreal, Canada, pp. 169–186.
[41] Richards, G., Hammer, C., Burg, B., Vitek, J., 2011. The eval that men do – a large-scale study of the use of eval in javascript applications. In: Mezini, M. (Ed.), ECOOP 2011 – Object-Oriented Programming. Vol. 6813 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 52–78.
[42] Rieck, K., Krueger, T., Dewald, A., December 2010. Cujo: efficient detection and prevention of drive-by-download attacks. In: Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10). Austin, Texas, USA, pp. 31–39.
[43] Schölkopf, B., Smola, A. J., 2002. Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. Adaptive Computation and Machine Learning. Mit Press.
[44] Seifert, C., Komisarczuk, P., Welch, I., June 2009. True positive cost curve: A cost-based evaluation method for high-interaction client honeypots. In: Proceedings of the 3rd International Conference on Emerging Security Information, Systems and Technologies (SECURWARE '09). Athens, Greece, pp. 63–69.
[45] Seifert, C., Welch, I., Komisarczuk, P., April 2007. Honeyc: The low-interaction client honeypot. In: Proceedings of the 5th NZ Computer Science Research Student Conference (NZCSRSC '07). Hamilton, New Zealand.
[46] Seifert, C., Welch, I., Komisarczuk, P., March 2008. Application of divide-and-conquer algorithm paradigm to improve the detection speed of high interaction client honeypots. In: Proceedings of the 23rd ACM symposium on Applied computing (SAC '08). Fortaleza, Ceara, Brazil, pp. 1426–1432.
[47] Seifert, C., Welch, I., Komisarczuk, P., December 2008. Identification of malicious web pages with static heuristics. In: Proceedings of the annual Australasian Telecommunication Networks and Applications Conference (ATNAC '10). Adelaide, Australia, pp. 91–96.
[48] Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., Boneh, D., 2004. On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM conference on Computer and communications security (CCS '04). Washington DC, USA, pp. 298–307.
[49] Sol, R., Guillon, C., Quintão Pereira, F., Bigonha, M., 2011. Dynamic elimination of overflow tests in a trace compiler. In: Compiler Construction. Springer, pp. 2–21.
[50] Sophos, December 2010. Why hackers have turned to malicious javascript attacks. Tech. rep.
[51] Stokes, J. W., Andersen, R., Seifert, C., Chellapilla, K., 2010. Webcop: Locating neighborhoods of malware on the web. In: Proceedings of the 3rd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More (LEET '10). San Jose, California, USA, pp. 5–13.
[52] Wang, Y., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., King, S., February 2006. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In: Proceeding of the 13th Annual Symposium on Network and Distributed System Security (NDSS'06). San Diego, California, USA.
[53] Warrender, C., Forrest, S., Pearlmutter, B., May 1999. Detecting intrusions using system calls: alternative data models. In: Proceedings of the 20th IEEE Symposium on Security and Privacy. Oakland, California, USA., pp. 133–145.
[54] Zhang, J., Seifert, C., Stokes, J. W., Lee, W., March 2011. ARROW: Generating signatures to detect drive-by downloads. In: Proceeding of the 20th International Conference on World wide web (WWW '11). Hyderabad, India, pp. 187–196.