Autonomous Learning for Detection of JavaScript Attacks: Vision or Reality?

Guido Schwenk
Technische Universität Berlin
Berlin, Germany

Alexander Bikadorov
Technische Universität Berlin
Berlin, Germany

Tammo Krueger
Technische Universität Berlin
Berlin, Germany

Konrad Rieck
University of Göttingen
Göttingen, Germany

ABSTRACT

Malicious JavaScript code in webpages is a pressing problem in the Internet. Classic security tools, such as anti-virus scanners, are hardly able to keep abreast of these attacks, as their obfuscation and complexity obstructs the manual generation of signatures. Recently, several methods have been proposed that combine JavaScript analysis with machine learning for automatically generating detection models. However, it is open how these methods can really operate autonomously and update detection models without manual intervention. In this paper, we present an empirical study of a fully automated system for collecting, analyzing and detecting malicious JavaScript code. The system is evaluated on a dataset of 3.4 million benign and 8,282 malicious webpages, which has been collected in a completely automated manner over a period of 5 months. The results of our study are mixed: For manually verified data excellent detection rates up to 93% are achievable, yet for fully automated learning only 67% of the malicious code is identified. We conclude that fully automated systems are still a vision and several challenges need to be solved first.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; I.5.1 [**Pattern Recognition**]: Models—*Statistical*

Keywords

Javascript Attacks, Anomalous Behavior Detection, Malware Identification

1. INTRODUCTION

According to a recent study of Symantec [26], the number of JavaScript attacks in the Internet has almost doubled in the last year, reaching peaks of over 35 million attacks per day. As part of these attacks, malicious JavaScript code is planted on webpages, such that a user visiting the webpage is automatically attacked and unnoticably infected with malicious software. The success of these attacks is rooted in the close interaction of the JavaScript interpreter with the web browser and its extensions. Often it is possible with a few lines of code to probe and exploit vulnerabilities in the browser environment [5, 7].

Unfortunately, the detection of malicious JavaScript code is a challenging task: JavaScript attacks are small programs that are executed in the web browser. The attacker can build on the full flexibility of interpreted code, which allows him to easily obfuscate his code as well as dynamically exploit different types of vulnerabilities. Common security tools, such as anti-virus scanners, are hardly able to keep abreast of these attacks, as the obfuscation and complexity obstruct the manual generation of effective signatures. As a result, malicious JavaScript code is often insufficiently detected due to a lack of up-to-date signatures [22].

As a remedy, several detection methods have been proposed that combine JavaScript analysis with techniques from the area of machine learning. These methods build on the ability of machine learning to automatically generate detection models from known samples of benign and malicious JavaScript code and thereby avoid the manual crafting of signatures. Common examples are the detection systems Cujo [22], Zozzle [4] and IceShield [10], which are capable of accurately identifying malicious code in webpages at run-time with few false alarms.

Learning-based detection provides a promising ground for mitigating the threat of malicious webpages. However, to take effect and provide advantages over signature-based tools, learning-based methods need to operate with very little manual intervention. From the acquisition of training data to the generation of detection models, the learning process needs to be largely automatic to quickly adapt to the development of malicious software. Previous work has ignored this issue of automatic learning and it is open whether learning-based detection methods can really operate autonomously over a longer period of time.

In this paper, we test the feasibility of automatic learning and present an empirical study of a fully automated system based on the detector Cujo [22]. The system (a) retrieves benign and malicious JavaScript code from the Internet, (b)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AISec'12, October 19, 2012, Raleigh, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1664-4/12/10 ...$15.00.

identifies malicious functionality using client-based honeypots and (c) learns a detection model from features of static and dynamic analysis in regular intervals. We evaluate the system on a dataset of 3.4 million benign and 8,282 malicious webpages, which has been acquired over a period of 5 months. In particular, we study the detection performance as well as the learning process over time for different features and learning methods, such as anomaly detection and classification approaches.

The results of our study are mixed: In line with previous work, the system attains a high detection rate of 93% if applied to manually verified data. However, in a fully automated setting it identifies only 67% of the malicious code in webpages—irrespective of the used features and learning methods. We identify two main factors that contribute to this decrease:

- *Semantic gaps:* It is considerably hard to verify the presence of malicious activity during the visit of a webpage and use the exact same information at a later stage for learning. If both stages differ only slightly, malicious activity may be present but is not exposed to the learning method.
- *Time delays:* JavaScript attacks are very volatile and often active for only a few hours. Due to the large amount of processed data, a significant amount of time may pass between the verification of a malicious webpage and the resulting learning stage. If the malicious code is not present anymore, the detection model is trained on incomplete data.

Overall, we conclude from our study that fully automated systems for detection of JavaScript attacks are still an open problem and there exist several practical challenges that need to be addressed first.

The rest of this paper is structured as follows: We first discuss related work in Section 2. Section 3 then introduces our framework for data acquisition and presents details of the collected JavaScript code. Section 4 describes the features and learning methods used in our system. Section 5 finally presents the results of our study and discusses their implications. Section 6 concludes.

2. RELATED WORK

Before presenting our study on learning-based detection of malicious JavaScript code, we first review some related work. In particular, we discuss related approaches for analyzing and detecting malicious code in webpages. These approaches can be roughly categorized into *client-based honeypots*, *analysis systems* and *detection systems*, where these categories are not rigid and some systems implement a mixture of functionalities.

2.1 Client-based Honeypots

To systematically monitor and understand the phenomena of JavaScript attacks, several honeypot systems have been devised that visit webpages and mimic the behavior of users. One class of these systems are high-interaction honeypots [e.g., 20, 23, 25, 27], which operate a real browser in a sandbox environment and detect attacks by monitoring unusual state changes in the environment, such as modified system files. Another class of these systems are low-interaction honeypots, which only emulate the functionality of web browsers and corresponding vulnerabilities for tracking malicious activity [e.g., 1, 11, 19].

Both types of honeypots are valuable sources for collecting JavaScript attacks, especially in combination with systems for efficient retrieval of potentially malicious webpages [12]. In contrast to server-based approaches, client-based honeypots are capable of actively searching for malicious code and allow to capture instances of novel attack campaigns early on. As a consequence, client-based honeypots are widely used and can be considered a standard for monitoring JavaScript attacks in the wild.

2.2 Analysis Systems

Collecting malicious JavaScript code, however, is only a first step in crafting effective defenses. A second strain of research has thus focused on methods for automatically analyzing the collected code and extracting security-relevant information, such as patterns indicative for attacks. Most notable here is the community service WEPAWET that is backed by a chain of analysis tools for collecting, filtering and analyzing JavaScript code [2, 3, 12]. The service automatically analyzes webpages using an emulated browser environment and is able to identify anomalous behavior in the code using machine learning techniques.

In contrast to WEPAWET, which performs an analysis of webpage content in general, other systems address particular aspects of JavaScript attacks [e.g., 13, 14]. For example, the analysis system ROZZLE implements an involved multipath execution for JavaScript code. Instead of following a single execution flow, the method inspects multiple branches of execution and thereby exposes hidden and conditional functionality of JavaScript attacks.

Although very effective in analyzing code and identifying JavaScript attacks, the presented analysis systems are mainly designed for offline application and induce an overhead which is prohibitive for real-time detection. For example, Cova et al. [3] report an average processing time of 25 seconds per webpage for WEPAWET. For this reason, we do not consider methods for offline analysis in our study—even if they employ learning-based components. Nevertheless, many of the techniques implemented for offline analysis are also applicable in online detection systems [14].

2.3 Attack-specific Detection

The first methods capable of detecting malicious code at run-time have been proposed for specific types of JavaScript attacks [e.g., 8, 21]. These methods proceed by monitoring the browser environment for known indicators of certain attack types. For example, the system NOZZLE scans string objects for fragments of executable code, a typical indication of heap-spraying and other memory corruption attacks. While these approaches provide a low run-time, they are inherently limited to particular attacks and do not provide a generic protection from malicious JavaScript code.

A more generic detection of JavaScript attacks is achieved by the systems BLADE [17] and ARROW [28], which identify attacks using indicators outside the browser environment. In particular, BLADE spots and blocks the covert installation of malware as part of drive-by downloads, whereas ARROW generates detection patterns for the URLs involved in JavaScript attacks. Both methods intentionally do not analyze JavaScript code and are thus independent of specific attack types. However, by ignoring the actual attack code,

these methods critically depend on the presence of the considered indicators in practice.

2.4 Learning-based Detection

The demand for a generic detection of malicious code has finally motivated the development of efficient learning-based detection systems, such as CUJO [22], ZOZZLE [4], and ICESHIELD [10], which are the main focus of our study. These systems analyze webpages at run-time and discriminate benign from malicious JavaScript code using machine learning techniques. In contrast to offline analysis, they induce only a minor run-time overhead and can be directly applied for protecting end user systems.

At the core of these learning-based approaches are two central concepts: the *considered features* and the *learning model* for detecting attacks. For example, ZOZZLE mainly extracts features from a static analysis of JavaScript code, whereas ICESHIELD monitors the execution of code dynamically and constructs behavioral features. Moreover, many efficient detection systems employ a supervised classification approach for learning, while the offline system WEPAWET successfully uses unsupervised anomaly detection for identifying attacks. We study these concepts and related differences in our evaluation in more detail later.

3. DATA ACQUISITION

A key for evaluating learning-based detection systems is a realistic dataset of malicious and benign JavaScript code. Previous work has suggested to automatically acquire such data using client-based honeypots and offline analysis systems. This is clearly a promising approach, as it allows for automatically updating and re-training learning-based systems on a regular basis. However, almost no research has explored this approach in depth. Most of the results reported for learning-based detection have been obtained on a single dataset with manually cleansed training data.

In this study, we investigate how learning-based systems perform if they are regularly and automatically updated with malicious and benign data *without* human sanitization. To this end, we have devised a framework that visits malicious and benign webpages on a daily basis and returns reports for static and dynamic analysis of the contained JavaScript code.

3.1 Collection Framework

An overview of the collection framework is presented in Figure 1. The framework is constructed using existing security instruments, such as public services and client-based honeypots, and only serves the purpose of automatically retrieving large amounts of benign and malicious JavaScript code. Note that the framework is not designed to gain insights into the malware ecosystem. Moreover, we deliberately exclude learning-based components from the framework, such as PROPHILER [2] and JSAND [3], as they may bias the evaluation of learning-based detection methods towards their specific feature sets.

3.1.1 Sources for URLs

At the start of each day sources of potentially benign and malicious URLs are harvested. For the benign URLs we consider rankings and listing of popular webpages. In particular, we randomly sample 25.000 URLs per day from the Alexa ranking, which lists the top 1 million web pages according to visitors and page views. Popular web pages are not necessarily attack-free. In fact, attackers invest considerable effort into comprising popular webpages and exposing malicious code to a large group of users. Consequently, we cannot rule out that some of the 25.000 URLs are compromised, yet we assume that the vast majority of the URLs is benign. Moreover, we take precautions in the later verification to filter out known instances of JavaScript attacks.

For collecting potentially malicious URLs, we visit common blacklists and services tracking malicious URLs. As an example, we query the database service HARMUR [16] for all malicious URLs that have been submitted in the last 24 hours. Furthermore, we regularly retrieve URLs from search engines using "dangerous" search terms. In total our framework collects about 8,000 potentially malicious URLs per day. Similarly to the benign sources, these URLs are not guaranteed to be malicious and thus the subsequent verification of the data is an indispensable step.

3.1.2 Verification

The collected benign and malicious URLs are far from being an ideal source for training and evaluating learning-based systems. First, the data is not guaranteed to be of a certain type and, second, even if a URL points to a malicious webpage, this does not necessarily indicate that JavaScript code is involved. Note that phishing and other scam webpages are also often flagged as malicious but do not contain any malicious JavaScript code. To improve the quality of our data, we thus employ a verification stage that filters the data, such that malicious and benign JavaScript code is obtained with high probability. For this task we focus on tools that could also be applied in a practical deployment scenario, that is, we conduct the verification on a single workstation and only use techniques that return results within a few hours.

We apply the following two verification procedures to benign and malicious URLs, respectively:

(a) For improving the quality of benign URLs, we filter out blacklisted URLs from the 25,000 URLs collected per day. To carry out this task efficiently, we use the Google Safe Browsing service that provides a frequently updated list of malicious URLs. The main goal of this stage is to remove known malicious code from the benign dataset.

(b) We analyze all collected malicious URLs using a high-interaction honeypot. In particular, we use the honeypot SHELIA [23] which monitors a target application and employs taint tracking for identifying memory corruption and code injection attacks. As target application we use the Internet Explorer 6.

As a result of this analysis, we can refine our collected data to webpages that are either (a) likely benign and not contained in blacklists or (b) likely malicious and cause memory corruption or redirection of control flow during execution. We focus on a particular setting, namely SHELIA and the Internet Explorer 6, as this browser version is known for several public vulnerabilities and a frequent target of attacks. Nevertheless, our framework could be easily extended to also support other browsers and client-based honeypots for the verification stage.

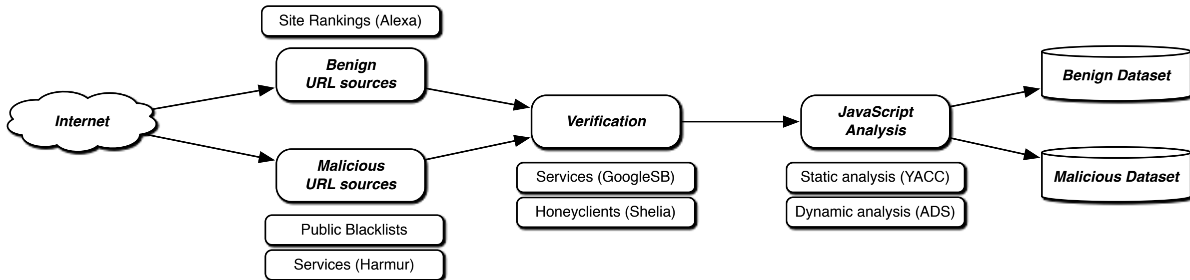

Figure 1: A framework for acquisition and analysis of JavaScript code.

3.1.3 Javascript Analysis

While learning-based detection systems generally follow the same design template, their inner working differs fundamentally. For example, ZOZZLE closely interacts with the JavaScript interpreter of the Internet Explorer, ICESHIELD uses frozen DOM objects for tracking JavaScript execution and CUJO makes use of a dedicated sandbox for analyzing code. Clearly, integrating all these different technical systems into a single prototype and conducting a fair comparison would be an intractable task.

Therefore we focus on a single system, namely CUJO, as it provides static and dynamic analysis of JavaScript code. While there are several subtle differences in how the code is analyzed, the reports of the static and dynamic analysis are basically similar to the representations of code used in the other systems. Overall, we are not interested in benchmarking concrete implementations but rather comparing underlying concepts, such as the use of static or dynamic code analysis for detection.

Static analysis. The static analysis in our framework first assembles the code base for a given URL by following redirects and downloading referenced JavaScript code. The assembled code is then parsed using a YACC grammar and the parsed representation is passed as a report for further analysis. A detailed description of the parsing process is presented by Rieck et al. [22].

Dynamic analysis. Additionally, a dynamic analysis of the JavaScript code is conducted. This analysis uses an enhanced version of ADSANDBOX (ADS), an efficient sandbox for JavaScript code. This sandbox is embedded in the Internet Explorer and allows to observe the behavior of JavaScript code in a secure environment. All interactions of the code with the virtual browser are recorded and a detailed report of the code's behavior is generated. A description of the sandbox is provided by Dewald et al. [6].

The presented framework enables us to automatically collect benign and malicious webpages and to generate analysis reports for each webpage on a daily basis. It is necessary to note that analyzing over 25.000 URLs and processing 8,000 URLs with a honeypot each day is a challenging task. As a consequence, several hours may pass during the processing of a webpage and a verified malicious URL may not necessarily expose malicious activity when it is later visited using the JavaScript analysis. This problem of *time delays* is inherent to our setting and would also exist in a practical application. Hence, we do not artificially correct our data and leave inactive attacks in our malicious dataset.

3.2 Collected Data Sets

The framework for data acquisition has been deployed on April 19th 2011 at our site and collected malicious and benign JavaScript code for a period of 5 months (137 days). Though we do not artificially correct our data and leave inactive attacks in our malicious dataset, the reports of ADSANDBOX are filtered to remove those that are broken, empty, timed out during the analysis or are too small to show any specific behavior, since such reports corrupt the training process with their inconsistent features. Table 1 shows the total number of webpages before and after filtering, and the resulting dataset sizes.

	Benign	Malicious
Total number of URLs	3,400,000	8,282
Number of filtered URLs	2,900,000	3,220
Total size of dataset	359,000 MB	130 MB
Size of filtered dataset	291,000 MB	77 MB
Unique filtered URLs	1,200,000	2,146

Table 1: Details of the acquired datasets.

As described in Section 3.1, initially 25.000 benign and 8,000 potentially malicious URLs have been collected per day. The applied verification significantly reduced the number of malicious URLs, as only a fraction of the candidate URLs has been capable to trigger an attack in our honeypot. Figure 2 depicts the number of malicious URLs that have been visited and verified per day.

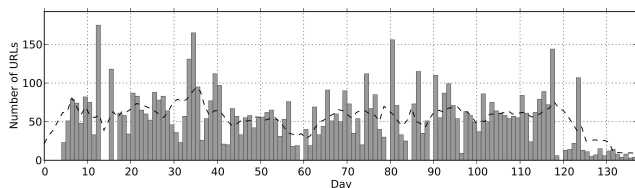

Figure 2: Number of malicious URLs visited per day. The dashed line displays a 7-day average.

The corresponding size of the downloaded JavaScript code per day for the benign and the malicious URLs, averaged for each day, is depicted in Figure 3. An interesting observation is the continuous growth of the amount of JavaScript code found at the benign URLs, reflecting the generally increasing importance of JavaScript. The peak at day 78 is due to a down-time of our analysis system and the resulting queue of unvisited webpages. For the malicious URLs a certain

kind of recurring structure seems to exist, showing a weekly periodicity in the peaks and average lines. We credit this finding to compromised workstation systems involved in the attacks, e.g. by redirecting traffic or hosting landing pages.

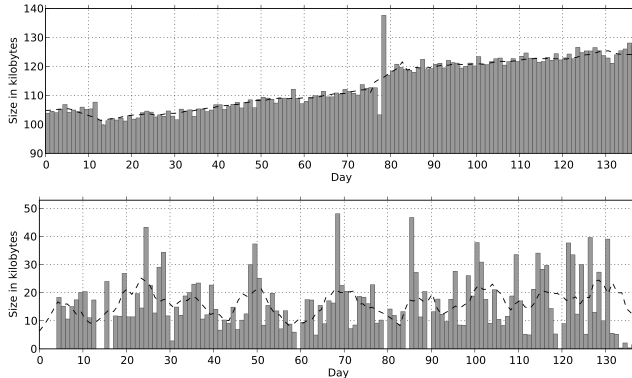

Figure 3: Average size of JavaScript code collected per day from benign (top) and malicious (bottom) URLs. The dashed line displays a weekly average.

4. LEARNING-BASED DETECTION

All learning methods heavily rely on the *features* chosen to model a certain problem. The features for the detection of malicious JavaScript code require the ability to reflect patterns in the code as well as monitored malicious behavior. Subsequently a proper *learning method* has to be selected, which is capable of handling those features. To overcome the limitation of choosing either a supervised or an unsupervised learning method, we discuss a way of evaluating both types of learning methods in a single setup. The following sections present this setup, the set of features and learning methods considered in our study, and the resulting learning framework.

4.1 Feature Extraction

We focus on features that can be extracted from the available static and dynamic reports. Those features are independent of specific attack characteristics, while reflecting relevant properties of the contained JavaScript behavior. In particular we apply *q-gram*-features to extract short string features from the reports, as implemented in CUJO. Moreover, we use specific numeric features derived from WEPAWET and ICESHIELD.

Q-gram features. Q-grams denote sequences of q words. To extract q-grams from a static or dynamic report, the report is transformed into a sequence of words, separated by white-space characters. The desired q-grams are extracted by sliding a window with the size of q words over the report and storing the consecutive q words found at each position. The following example depicts this procedure for a snippet of a static report, using a value of $q = 3$:

`ID = ID + x` ⇒ { `(ID = ID)`, `(= ID +)`, `(ID + x)` }.

The procedure works similar for the dynamic reports, where the `SET` command is executed on a variable `x.y`, leading to a similar list of sequences:

`SET x.y to "a"` ⇒ { `(SET x.y to)`, `(x.y to "a")` }.

In the next step the q-grams of each report are mapped into a vector space spanned by all possible q-grams, that is, each dimension is associated with the occurrences of one particular q-gram. This results in a high-dimensional, yet sparse vector for each report, which can be efficiently processed [22].

Numeric features. Additionally, we derive 15 different numeric features by analyzing the static and dynamic reports, similar to the way such features are extracted in WEPAWET and ICESHIELD. These features are designed bearing specific malicious JavaScript behavior in mind. Exemplary features inspired by WEPAWET are counts of the occurrences of `document.location` or `document.referrer`, the number of instantiated components, the number of times code is executed dynamically or the ratio of string definitions and string uses. Examples of features inspired by ICESHIELD are the number of occurrences of dynamically injected code, occurrences of potentially dangerous MIME-types or abnormally long strings used in decoding functions.

Unfortunately we could not accurately re-produce the features of WEPAWET and ICESHIELD, as some of them require information which is not contained in our analysis reports. Thus the combination of this incomplete feature set and our heterogeneous data leads to an insufficient detection performance for these numeric features. To avoid a misleading presentation of results, we therefore decided to omit numeric features in the following experiments.

4.2 Learning-Based Detectors

Taking insights from CUJO, the behavior of different learning models is studied using *Support Vector Machines* (SVM) [18, 24] with linear kernels. Though other approaches like decision trees work as well, SVMs offer a high performance and are robust against noise in the data. The feature space depends on the choice of the kernel. Focusing specifically on SVMs with a *linear kernel* has two advantages. The first one is the ability to process big datasets of very high dimensionality very fast. While Gaussian kernels often lead to better results than using a linear kernel, this comes at a massively increased cost of run-time and memory requirements. The second advantage is the opportunity to parametrically balance the influence of differently sized datasets during training. Before discussing this in more detail, some conceptual basics are covered.

Basics of two-class SVMs. A two-class SVM model is trained on datasets of two classes. Objective of the training is to find a hyperplane between the two classes which separates them with a maximal margin. The data points of each class with feature vectors closest to this margin are denoted as *support vectors*. In our setting, those two classes correspond to the reports retrieved from the benign and the malicious URLs. For each report the feature vectors are extracted. Once the model is trained on the feature vectors of those two datasets, an unknown report can be classified very fast. For this purpose the feature vector of this unknown report is extracted. Afterwards its position and distance from the hyperplane allows to predict the corresponding class membership. Figure 4 depicts an example of benign (white) and malicious (black) data points, where different models have been trained on. The dashed line of the middle image represents the separating hyperplane.

Two-class learning methods work best if the two classes have a comparable size. If one of the classes is much bigger,

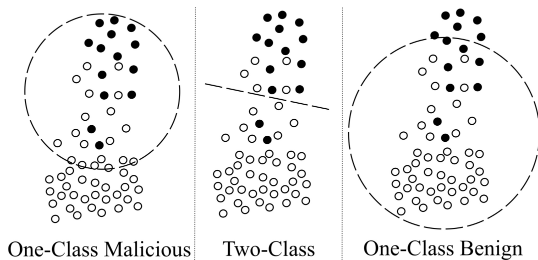

Figure 4: Visualization of different models achieved by different values of ω.

however, this approach often leads to suboptimal models. In that case one-class learning is often the better approach, allowing to learn a model on a huge dataset of one class alone.

Transition to one-class SVMs. Two-class and one-class learning methods both have their advantages and shortcomings. Focusing only on unsupervised one-class methods means ignoring our available malicious data, which makes calibrating the detector much harder. Focussing only on the supervised two-class methods, however, leaves the problem of the imbalanced size of both datasets. To achieve a smooth transition between both methods, we include a parameter ω into the model selection phase of the learning method. A high value of ω increases the weight on one specific class, which means that the two-class SVM operates like a one-class SVM for this class. This is done by applying a higher penalty for misclassifications of this class. It is also possible to choose ω such that it balances two differently sized classes. An example of the models resulting in using three different values of ω is illustrated in Figure 4 on a toy data distribution.

The utilization of ω during model selection facilitates a direct comparison of the detection performance of a two-class learning method with both a benign one-class learner and a malicious one-class learner, simply by applying different values of ω during model selection. A hope is that a properly defined ω could lead to a model better adapted to the imbalance of the two classes. As a result, neither the one-class nor the two-class models would be expected to be the optimal solution, but instead an optimized model in between.

4.3 Learning Framework

The long-term dataset acquired during this study, as well as the different selected features and learning methods allow for an extensive experimental evaluation. This whole evaluation is conducted in an offline manner under utilization of the SVM library LibLinear [9] for training the model. The extraction of *q-gram* features from the static and the dynamic reports leads to a static and a dynamic detector, respectively. Taking insights from Cujo, we fix the parameter q to $q = 4$ for the static reports and to $q = 3$ for the dynamic reports.

To evaluate the behavior of the different detectors during the whole time period of the study, we train and test our system weekly, i.e. every seven days all models are re-trained. To cope with the circumstance that the number of malicious reports is much smaller than the number of available benign reports, all past malicious reports are used during the training phase, but only the last two weeks of benign reports are used. We split these two week dataset in half, such that the newest reports constitute the *training data*, while the older reports constitute the *validation data*. On these datasets we perform a model selection for both the static and dynamic models over the cost parameter c and the parameter ω. Then for each model the hyperplane is calibrated as follows: First the false positive rate of the current model is calculated on the validation data. Then the hyperplane is shifted such that a preselected false positive rate on the validation data, $FP_{val}(\Theta)$, defined by a threshold Θ, is not exceeded. Finally the model with the highest true positive rate on the validation data is selected.

Being able to define Θ is a vital component of training a learning based detector, suitable for the network environment at hand, because some environments simply require lower false positive rates than others. When testing the best model this targeted false positive rate is achieved with only minor deviations. Consequently no achieved false positive rates are shown in the evaluation, as they are close to the target values defined using the parameter Θ.

In our practical evaluation we pick a value of $FP_{val}(\Theta) = 0.001$ (i.e. 0.1%) as a good compromise of low false positive and high true positive rate. For some experiments, however, we also consider $FP_{val}(\Theta) = 0.0001$ to highlight specific properties. For convenience the Θ with $FP_{val}(\Theta) = 0.001$ is further on denoted as $\Theta_{0.001}$, and Θ with $FP_{val}(\Theta) = 0.0001$ is denoted as $\Theta_{0.0001}$. Note that the combination of the collection framework of Section 3.1 and this learning framework realizes a fully automated system for collecting, analyzing and detecting malicious JavaScript code.

5. EXPERIMENTS

In the evaluation we first investigate, whether learning-based detectors can easily be integrated in a completely automated tool-chain without manually sanitized data. This is empirically supported by a long-term evaluation of the performance of the different detectors and features utilized in our learning framework. Aided by this long-term evaluation the second part discusses the influence of different re-training procedures on the performance of the different detectors.

5.1 General Performance Evaluation

The following sections analyze the feasibility of learning-based detectors for the application in a completely automated framework. First of all the detection performance of two commonly used anti-virus tools is determined on our datasets, which is compared to the performance of our learning framework. In the next step the performance of our learning framework on a sanitized dataset is shown. After that examples and reasons for misclassified reports and the influence of ω are discussed.

5.1.1 Performance of AV-Scanners

The two anti-virus tools Avira Antirvir and AVG Anti-Virus have been tested on the original JavaScript code of our complete benign and malicious datasets. Both employ different analysis engines capable of detecting maliciously behaving JavaScript code. Table 2 shows the results of this analysis.

The achieved true positive rates are quite low. This result comes as a surprise, considering the extensive initial verification steps of our system, and considering the circumstance

JavaScript Code		Avira Antivir	AVG Anti-Virus
Benign URLs	FP	0.0007	0.0003
Malicious URLs	TP	0.2760	0.3140

Table 2: False positive and true positive rates of two anti-virus tools.

that the anti-virus tools have been applied several months after the last day of data collection, which gave the anti-virus vendors enough time to update their signatures. The achieved false positive rates range between the targeted false positive rates of $FP_{val}(\Theta_{0.001})$ and $FP_{val}(\Theta_{0.0001})$. This permits an easy comparison of the corresponding true positive rates to those achieved by our detectors.

5.1.2 Performance of Detectors

Additional to the performance of the individual static and dynamic detectors, the performance of a disjunctive combination of both detectors is evaluated as well. This combined detector triggers an alarm if either the static or the dynamic detector does so. The resulting average performance values of the different detectors, obtained from weekly re-trained models tested on the following week, are listed in Table 3.

Detector	$\Theta_{0.0001}$	$\Theta_{0.001}$
Static	0.3525	0.5409
Dynamic	0.4931	0.6208
Combined	0.5451	0.6733

Table 3: Average detection performance of the different detectors.

While the static and dynamic detectors are nearly on par, the combination of both is significantly higher. In comparison to the performance of the anti-virus tools, all detectors show a good detection performance. Especially the combined detector is able to classify approximately twice as much malicious URLs correctly. Surprisingly however, none of the considered features or learning methods attains a detection rate of more than 90%, as reported from previous work, which used manually sanitized datasets. To investigate this further we decide to create a subset of the malicious JavaScript code using the anti-virus tools as an additional sanitization instance. The assumption is that previous work always used manually sanitized datasets, so this way of automatic sanitization is expected to result in a better detection performance. In this new *AV-Alerts* dataset only those 890 URLs are included which both anti-virus tools raised an alert for. The results of the evaluation of our learning-based detection methods on this dataset are listed in Table 4.

Detector	$\Theta_{0.0001}$	$\Theta_{0.001}$
Static	0.7264	0.8446
Dynamic	0.7394	0.8480
Combined	0.8450	0.9319

Table 4: Average true positive rates of the different detectors, tested on the AV-alerts.

Especially the combined detector shows an impressively increased detection rate for both values of Θ, reaching a performance much closer to that of methods which are solely tested on manually sanitized data. This illustrates that – under the assumption of a sanitized dataset – the results of previous papers can be reproduced.

The application in a completely automated system, however, drastically decreases the detection performance. We see two main reasons for this unexpected behavior. While the verification phase of the collection framework acts properly in defining, whether the scripts of a potentially malicious URL really behave maliciously or not, we can not be certain that the specific malicious behavior SHELIA detected is also detected and reported by ADSANDBOX. Because the learned model builds on those reports of ADSANDBOX, it learns features that do not contain the initial malicious behavior any longer. We denote this as the *semantic gap*, because it is caused by the discrepancy of detection and learning mechanisms. The second reason is of a temporal kind. Because JavaScript attacks are very volatile and often active for only a few hours, any delay between the verification step and the creation of the reports may lead to a worse learned model, because the attack may already be inactive again and the malicious code not present anymore. We denote this as *time delays*.

5.1.3 Misclassification Analysis

There exist different reasons for misclassifications. One general reason is that the datasets are noisy, i.e. not all members of a class behave in accordance to their label. For the benign reports this means that, as explained in Section 3.1.2, the verification phase for the JavaScript code of benign URLs does not guarantee to exclude all malicious JavaScript code. Due to the *semantic gap* this is even more complex for the reports of malicious URLs. SHELIA might respond to a type of malicious behavior, which the static or dynamic reports of ADSandbox do not reflect. As a result the dynamic and static reports look benign. The dynamic reports might even be empty or have completely failed to execute. To test this last assumption, the malicious dataset has been further filtered, leaving only those 2,179 malicious URLs which did not result in errors in the dynamic execution. When testing the dynamic detectors on this dataset, the average detection performance increased only very slightly (approximately 1.5%), meaning that these errors in the dynamic execution do not influence the quality of the reports significantly.

To get a better idea of the concrete reasons for misclassifications in our system, we analyze the false positives and the false negatives that occurred using a representative model on the complete dataset. The histograms of the corresponding predicted scores of the static and the dynamic detectors, as well as the concrete values of $\Theta_{0.001}$, are illustrated in Figure 5. As a result we find that many of the false positives actually are malicious. Concrete examples of code injection, the dynamic execution of long obfuscated sequences, redirects to suspicious websites, hidden iframes and heapspraying occurred. Those false positives that have really shown a benign behavior often contained features similar to malicious features, e.g. dynamic execution of obfuscated code or hidden iframes, which renders a proper classification difficult.

The verification process of the malicious URLs is more elaborated, especially due to the incorporation of SHELIA. Therefore none of the false negatives is expected to contain benign JavaScript code. For this reason it is an indicator of the *semantic gap* that many samples of the false nega-

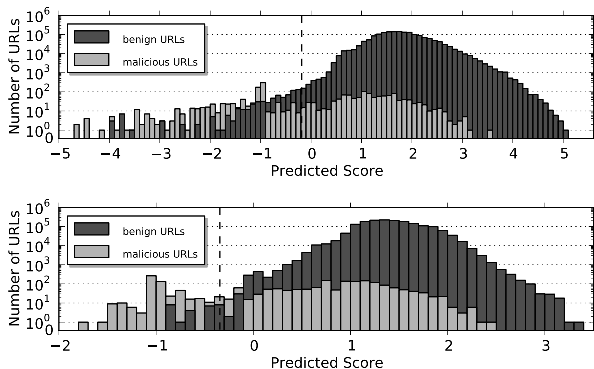

Figure 5: Histograms of the predicted scores of the optimized models of the static (top) and dynamic (bottom) detectors on the complete dataset. The dashed line represents $\Theta_{0.001}$.

tives did not expose any behavior at all. Other misclassifications have been caused by malicious code looking very similar to benign code, i.e. omitting features usually found in malicious reports, like heapspraying, code injection or the dynamic execution of obfuscated code.

5.1.4 Performance of Different Learning Methods

Another interesting question is the impact of the class-weight parameter ω on the performance of the different detectors. Specifically, whether the detection performance of the normal two-class model can be optimized by this step, and how the different one-class approaches perform. The results are listed in Table 5. The low performance of the different one-class models does not come as a surprise. The learned model simply can not rely on one class only, and the calibration does not work that well either. The detection performance of the two-class model is boosted by the use of the optimized ω, resulting in much better results than those of the generic two-class model. The optimal models of our evaluation always use values of ω corresponding to a model in between a benign one-class model and a two-class model. An analysis of the development of the optimal values of ω during the re-training setup also shows a steady shift of ω towards the benign one-class model, caused by the continuously growing number of malicious data points.

	$\Theta_{0.001}$	
Learning Models	Static	Dynamic
One-Class Benign	0.0375	0.0303
One-Class Malicious	0.1314	0.1757
Two-Class	0.4560	0.4794
Two-Class Optimized	0.5409	0.6208

Table 5: Average true positive rates of the different models.

Integrating a class weighting into the training phase of a detector to balance differently sized datasets improves the overall performance. The insight that the chosen optimal ω resides closer to a benign one-class learning model is helpful as well, because the benign one-class detector is due to the stability of its database, i.e. the better availability of benign data, more desirable than a model trained on continuously changing malicious data.

5.2 Analysis of Re-Training Procedures

The functionality and quantity of JavaScript code in real-life websites grows steadily, as we can see in Figure 3. A model should reflect this, because a model trained only once might be out-dated very soon. Thus a learning-based detector which is regularly re-trained on the latest datasets is assumed to achieve better models. The biggest disadvantage of this approach is the continuous requirement to regularly spend time and resources to compute an updated detector. While such learning efforts can be minimized, e.g. by using incremental learning [15], a confirmation of the assumed advantage of regular re-training in the domain at hand has yet to be done. For this purpose the following sections focus on a comparison of the long-term performance of the different detectors, either utilizing frequent re-training or conducting a one-time training only.

5.2.1 Regular Re-Training

In the re-training setup, the detectors are re-trained each week and tested on the following week. The corresponding overall average performance results have been discussed in Section 5.1.2. In Figures 6, 7, 8 and 9 the weekly average is represented by the dashed line. The light vertical bars contained in the figures visualize days where for none of the 8,000 verified malicious URLs an actual malicious behavior could be exposed.

Figure 6 illustrates the performance values of the static detector. A first observation is the immense variance of both false and true positive rates. The picture is slightly different for the dynamic detector, illustrated in Figure 7. Its false positive rates show much less variance, while its true positive rates fluctuate massively, even more than the ones of the static detector. An especially interesting observation is, that the true positive rates of both the static and dynamic detector are weak in the beginning, while generally improving the average performance afterwards. This is caused by the low number of malicious data available for training at the very beginning of the evaluation. With the steadily increasing number of malicious reports available during training, however, better models are achieved. Another interesting observation is the performance drop of the models closely following day 78. At that day a huge amount of URLs has been updated after a down-time of the collection framework, which had an impact on the dynamic detector, but not the static detector.

The long-term performance of the combined detector, depicted in Figure 8, is more stable than the one of the individual detectors. Especially the periods of low performance of the dynamic detector are nicely backed up by the static detector. The general false positive rate has doubled as well, but the combination of the detectors still performs best.

5.2.2 One-Time Training

In the one-time training setup a single model is trained on both the static and the dynamic reports, respectively. Because a sufficient amount of malicious data is vital for achieving a good detection performance, the model is not trained on the first weeks, but on the seventh, where a sufficient amount of malicious data is finally available. These models are then tested on all consecutive days. Note that for

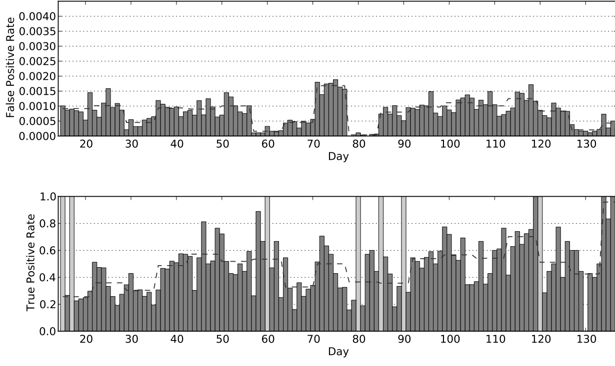

Figure 6: From top to bottom: False and true positive rates of the static detector per day with regular re-training, using $\Theta_{0.001}$.

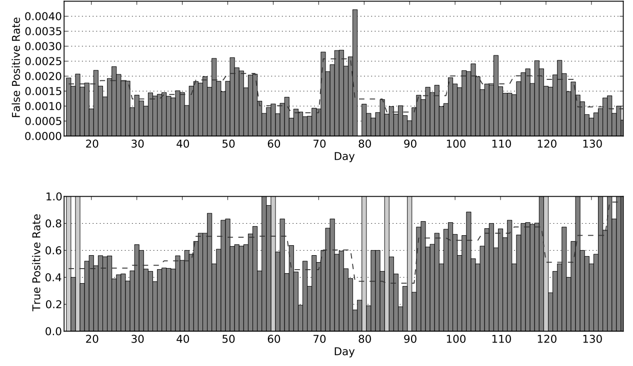

Figure 8: From top to bottom: False and true positive rates of the combined detectors per day with regular re-training, using $\Theta_{0.001}$.

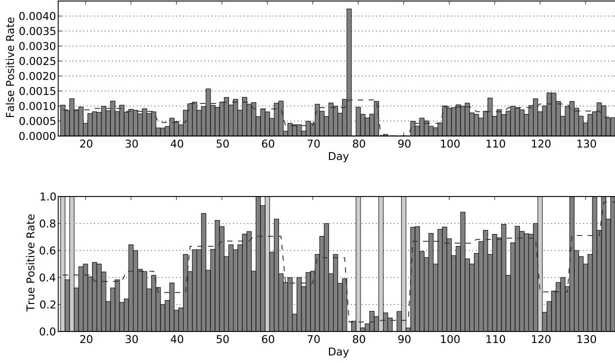

Figure 7: From top to bottom: False and true positive rates of the dynamic detector per day with regular re-training, using $\Theta_{0.001}$.

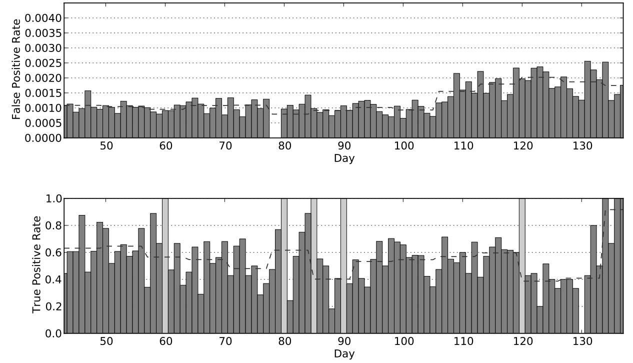

Figure 9: False and true positive rates of the dynamic detector per day, in the one-time training setup, using $\Theta_{0.001}$.

reasons of comparability the average detection performances of the previous tables are calculated on this range as well.

The dynamic detector exposes some interesting properties when comparing its re-training performance, depicted in Figure 7, with the corresponding one-time performance, depicted in Figure 9. The re-trained models show a much more varying performance than the model which has been trained only once. Especially the true positive rate is much more stable. The implication for the dynamic detector is, that a re-trained model is not necessarily better than an existing one. The static detector does not expose such varying behavior.

In terms of the long-term detection performance during the one-time training setup, an initial expectation was that applying an old model on newer data could lead to a steady performance decrease. This could not be observed for the static detector, where the false positive rate remained very stable and even the true positive rates follow closely the development curves observed in the re-training setup. The performance of the dynamic detector in Figure 9 does comply with that initial expectation a little more. The false positive rate, while stable for quite some time, starts to increase from day 100, and the true positive rate is below the average for nearly two weeks at that time. These results suggest that using the detectors alone for such a long time without re-training is no viable option.

Detector	$\Theta_{0.0001}$	$\Theta_{0.001}$
Static	0.2444	0.5027
Dynamic	0.3371	0.5531

Table 6: Average true positive rates of the different detectors applied during one-time training.

Table 6 finally shows the average results during the evaluation of the one-time training period. The performance values are generally lower than those of the re-training experiments, especially for $\Theta_{0.0001}$ (see Table 3 for comparison). The implications for the static detector are that while the detection performance does not decrease or vary that much during the one-time training setup, its average performance is considerably lower than the one achieved with regularly re-trained models. The average performance of the dynamic detector also suffers, but as discussed above, the lower variance of the detection performance of the dynamic detector makes a less frequent re-training of this detector a viable option.

6. CONCLUSION

In this paper we have investigated the feasibility to combine a learning-based system for the detection of malicious JavaScript code with a completely automated system for the collection and analysis of JavaScript code. The behavior and detection performance of different learning-based detectors, based upon different features and learning methods, have been evaluated on a huge set of automatically collected data, where previous work solely used manually sanitized datasets.

The results of this evaluation have shown that the vision of a complete, automated, learning-based system has not been achieved. Two main factors have been identified as contributors to these results. The first reason is the *semantic gap* occurring when malicious activity is present during the visit and verification of a webpage but no longer during the subsequent learning stage. Already a slight discrepancy in this information transfer means that malicious activity may be present but is not exposed to the learning method. The second reason is the *time delay*, caused by the volatility of JavaScript attacks which are often active for only a few hours. Due to the large amount of data accumulated and processed during the visit and verification of the URLs, a significant amount of time may pass between the verification of a malicious webpage and the resulting learning stage. Consequently the malicious code may not be present anymore, which results in a sub-optimal detection model, because it is trained on incomplete data.

Fortunately these problems can be solved by creating an integrated system which combines the components of collection, analysis and detection very closely. As a result the complete behavior, exposed during the collection and verification of the malicious URLs, should be available to the learning component with a minimum time delay.

Furthermore the evaluation has shown that better models can be learned by integrating the class-specific weight ω in the model selection and taking care of a sufficient amount of malicious data. Combining different detectors is also important, because they often supplement each other. And finally it has been shown that a regular re-training mostly results in a better detection performance than using a single model for a longer time period.

To bring autonomous learning to reality, a critical step is the design and development of an integrated analysis and learning system. Besides that another important idea is the investigation of a more intelligent way of re-training the different models based on their comparative performance. For example one could re-train one detector regularly, but instead of relying completely on the newly trained detector, just use it in parallel to the current one. Also methods of online learning are an interesting option for that purpose.

7. ACKNOWLEDGEMENTS

The authors would like to thank Andreas Dewald for providing a modified version of ADSandbox as well as Marco Cova for providing a set of JavaScript attacks. The authors gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) under the project PROSEC (FKZ 01BY1145).

References

[1] A. Büscher, M. Meier, and R. Benzmüller. Throwing a monkeywrench into web attackers plans. In *Proc. of Communications and Multimedia Security (CMS)*, pages 28–39, 2010.

[2] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proc. of the International World Wide Web Conference (WWW)*, pages 197–206, Apr. 2011.

[3] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proc. of the International World Wide Web Conference (WWW)*, 2010.

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

[5] M. Daniel, J. Honoroff, and C. Miller. Engineering heap overflow exploits with JavaScript. In *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*, 2008.

[6] A. Dewald, T. Holz, and F. Freiling. Adsandbox: Sandboxing JavaScript to fight malicious websites. In *Proc. of ACM Symposium on Applied Computing (SAC)*, pages 1859–1864, 2010.

[7] M. Egele, E. Kirda, and C. Kruegel. Mitigating drive-by download attacks: Challenges and open problems. In *Proc. of Open Research Problems in Network Security Workshop (iNetSec)*, 2009.

[8] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 88–106, 2009.

[9] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research (JMLR)*, 9:1871–1874, 2008.

[10] M. Heiderich, T. Frosch, and T. Holz. IceShield: Detection and mitigiation of malicious websites with a frozen dom. In *Recent Adances in Intrusion Detection (RAID)*, Sept. 2011.

[11] A. Ikinci, T. Holz, and F. Freiling. Monkey-spider: Detecting malicious websites with low-interaction honeyclients. In *Proc. of Conference "Sicherheit, Schutz und Zuverlässigkeit" (SICHERHEIT)*, pages 891–898, 2008.

[12] L. Invernizzi, S. Benvenuti, P. M. Comparetti, M. Cova, C. Kruegel, and G. Vigna. EVILSEED: A guided approach to finding malicious web pages. In *Proc. of IEEE Symposium on Security and Privacy*, May 2012.

[13] S. Karanth, S. Laxman, P. Naldurg, R. Venkatesan, J. Lambert, and J. Shin. ZDVUE: Prioritization of javascript attacks to surface new vulnerabilities. In *Proc. of CCS Workshop on Security and Artificial Intelligence (AISEC)*, Oct. 2011.

[14] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: de-cloaking internet malware. In *Proc. of IEEE Symposium on Security and Privacy*, May 2012.

[15] P. Laskov, C. Gehl, S. Krüger, and K. R. Müller. Incremental support vector learning: Analysis, implementation and applications. *Journal of Machine Learning Research*, 7:1909–1936, Sept. 2006.

[16] C. Leita and M. Cova. HARMUR: Storing and analyzing historic data on malicious domains. In *Proc. of Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, Apr. 2011.

[17] L. Lu, V. Yegneswaran, P. A. Porras, and W. Lee. BLADE: An attack-agnostic approach for preventing drive-by malware infections. In *Proc. of Conference on Computer and Communications Security (CCS)*, pages 440–450, Oct. 2010.

[18] K.-R. Müller, S. Mika, G. Rätsch, K. Tsuda, and B. Schölkopf. An introduction to kernel-based learning algorithms. *IEEE Neural Networks*, 12(2):181–201, May 2001.

[19] J. Nazario. A virtual client honeypot. In *Proc. of USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[20] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *Proc. of USENIX Security Symposium*, 2008.

[21] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. Technical Report MSR-TR-2008-176, Microsoft Research, 2008.

[22] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *26th Annual Computer Security Applications Conference (ACSAC)*, pages 31–39, Dec. 2010.

[23] J. R. Roaspana. SHELIA: a client honeypot for client-side attack detection. Master's thesis, Vrije Universiteit Amsterdam, 2007.

[24] B. Schölkopf and A. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.

[25] C. Seifert and R. Steenson. Capture – honeypot client (Capture-HPC). Victoria University of Wellington, NZ, `https://projects.honeynet.org/capture-hpc`, 2006.

[26] Symantec Internet Security Threat Report: Trends for 2010. Vol. 16, Symantec, Inc., 2011.

[27] 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 *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2006.

[28] J. Zhang, C. Seifert, J. W. Stokes, and W. Lee. ARROW: GenerAting SignatuRes to Detect DRive-By DOWnloads. In *Proc. of the International World Wide Web Conference (WWW)*, pages 187–196, 2011.