Intelligent Defense against Malicious JavaScript Code

Tammo Krueger[1] and Konrad Rieck[2]
[1] Technische Universität Berlin, Germany
[2] University of Göttingen, Germany

Abstract

JavaScript is a popular scripting language for creating dynamic and interactive web pages. Unfortunately, JavaScript also provides the ground for web-based attacks that exploit vulnerabilities in web browsers and unnoticeably infect users with malicious software. Regular security tools, such as anti-virus scanners, increasingly fail to fend off this threat, as they are unable to cope with the rapidly evolving diversity and obfuscation of these JavaScript attacks.

In this article, we present Cujo, a learning-based system for detection and prevention of JavaScript attacks. Embedded in a web proxy, Cujo transparently inspects web pages and blocks the delivery of malicious JavaScript code. A lightweight static and dynamic analysis is performed, which enables learning and detecting malicious patterns in the structure and behavior of JavaScript code. To operate the system in practice we introduce an architecture for automatically collecting and sanitizing data for retraining Cujo. We demonstrate the efficacy of this architecture in an empirical evaluation, where Cujo identifies 93% of real attacks with few false alarms—even if the attacks are present in benign web pages during training of the system.

1 Introduction

The JavaScript language is widely used for creating dynamic and interactive web pages. The vast majority of popular web services, such as Google, Facebook and Twitter, make heavy use of JavaScript for presenting dynamic content and interacting with the user. In contrast to served-based scripting, JavaScript code is interpreted in the web browser of the user and allows for directly interfacing with the document object model and the browser environment. For example, JavaScript is regularly used for animation of objects, validation of user input, and asynchronous communication.

The versatility of JavaScript, however, comes at a price. JavaScript is increasingly used as part of web-based attacks that exploit vulnerabilities in browsers and infect users with malicious software. According to a recent study of Symantec [28], the number of such attacks has almost doubled in the last year, reaching peaks of over 35 million attacks per day. It is the tight integration of JavaScript with the browser that enables web-based attacks to easily probe and unnoticeably exploit vulnerabilities during the visit of a web page. Due to the complexity of browsers and their extensions, there exist numerous of such vulnerabilities ranging from insecure interfaces of third-party extensions to buffer overflows and memory corruption vulnerabilities [see 8, 10].

As a consequence of this development, the detection and mitigation of malicious JavaScript code has gained a focus in security research. Several approaches have been devised for spotting malicious activity in web pages, for example, using code emulation [11, 16, 22], sandboxing [9, 20], or web-based honeypots [19, 27]. Most of these approaches rely on heuristics and manually crafted rules and thus lack the ability to adapt to the rapidly changing threat of web-based attacks. This shortcoming has recently been addressed by combining JavaScript analysis with techniques from machine learning, which enables learning detectors for malicious JavaScript code *automatically* [5, 7, 13, 24].

In this article, we present one of the first of these learning-based detectors: Cujo [24]. The detector is embedded in a web proxy, where it transparently inspects the JavaScript code of web pages and blocks the delivery of malicious content. To this end, a lightweight static and dynamic analysis is performed, which enables learning and detecting malicious patterns in the structure and behavior of JavaScript code. To operate this system in practice we introduce an architecture for automatic retraining that collects and sanitizes training data automatically. This architecture enables one to adapt the detectors to novel attacks and provides a crucial component for keeping abreast of attack development.

We demonstrate the efficacy of Cujo and the training architecture in an empirical evaluation with 200,000 web pages and 600 real attacks. In this evaluation, Cujo outperforms related approaches and—with Zozzle [7]—provides the most accurate detection of malicious JavaScript code. In particular, Cujo identifies 95% of the attacks with a false-positive rate of 0.002%, corresponding to 2 false alarms in 100,000 visited web sites. Furthermore, the proposed architecture allows Cujo to be trained on unclean data, where it still identifies over 93% of the attacks—although half of these are present in benign web pages during training.

This article is organized as follows: We describe Cujo in Section 2 and introduce the training architecture in Section 3. An empirical evaluation is presented in Section 4. Related work is discussed in Section 5 and Section 6 concludes.

[0]This is a preprint of an article published in *Praxis der Informationsverarbeitung und Kommunikation (PIK) Vol. 35(1), de Gruyter, 2012.*

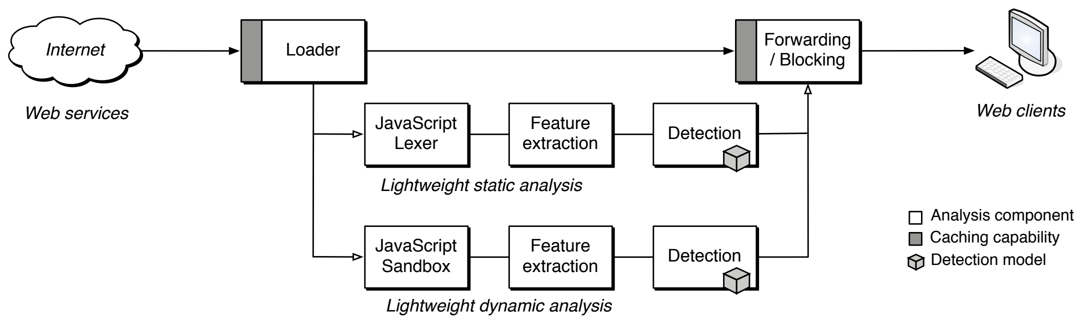

Figure 1: A schematic depiction of the Cujo detector [24].

2 The Cujo Detector

JavaScript attacks can take almost arbitrary form and structure depending on the exploited vulnerability and the use of obfuscation. For accurately detecting such attacks, a holistic view on JavaScript is required that takes into account structural as well as behavioral aspects of the code. In this section we provide a brief overview of the Cujo detector that combines a lightweight static and dynamic analysis of JavaScript code for detecting attacks. A detailed discussion of Cujo and its inner working is given by Rieck et al. [24].

Cujo is designed to prevent the delivery of malicious JavaScript code to a web client. Hence, Cujo is embedded in a web proxy, where it transparently inspects every requested web page and recursively downloads associated JavaScript code. Before any data is transmitted to the user, the code is analyzed and matched against static and dynamic detection models. If this analysis reveals any suspicious structure or behavior, the delivery of the web page is blocked. To avoid manually crafting detection rules for this analysis Cujo makes use of machine learning techniques, which enable generalizing from known attacks and allow to automatically construct detection models. A schematic depiction of the resulting system is presented in Figure 1.

2.1 JavaScript Analysis

The *static analysis* of Cujo relies on basic principles of compiler design [see 1]: Before source code written in a programming language can be processed, its textual representation has to be decomposed into tokens—a process referred to as lexical analysis. The static analysis in Cujo takes advantage of this process and parses the JavaScript code of a web page into lexical tokens. As a result, the code is represented as a sequence of tokens that capture the syntax and structure of the code.

The static analysis is illustrated in Figure 2 and Figure 3, where Figure 2 shows a snippet of obfuscated JavaScript code and Figure 3 the extracted lexical tokens. Note that the static analysis discards concrete identifiers, such as variable and function names. Instead only the syntax and structure of the code, such as the string operations and the for-loop, are extracted for further analysis.

```
a = "";
b = "{@xqhvfdsh+%(x<3<3%,>zk"+
    "loh+{1ohqjwk?4333,{.@{>";
for (i = 0; i < b.length; i++) {
    c = b.charCodeAt(i) - 3;
    a += String.fromCharCode(c);
}
eval(a);
```

Figure 2: Example of obfuscated JavaScript code.

```
ID = STR.000 ;
ID = STR.002 +
     STR.002 ;
FOR ( ID = NUM ; ID < ID . ID ; ID ++ ) {
    ID = ID . ID ( ID ) - NUM ;
    ID + = ID . ID ( ID ) ;
}
EVAL ( ID ) ;
```

Figure 3: Example of static analysis report.

Additionally to the static analysis, Cujo performs a *dynamic analysis* of JavaScript code that allows for monitoring and inspecting the behavior of a web page. To this end, Cujo uses an enhanced version of ADSandbox, a highly efficient JavaScript sandbox developed by Dewald et al. [9]. This sandbox emulates a virtual web browser and allows it to observe the behavior of JavaScript code in a secure environments. All interactions of the code with the virtual browser are recorded and a detailed report of the code's behavior is generated. In comparison to the static analysis, this report describes the actual behavior of the JavaScript code, irrespective of its structure and syntax.

As an example of the dynamic analysis, Figure 4 shows the behavior report obtained for the code snippet from Figure 2. The report includes all operations of the code that alter the environment of the virtual web browser, such as different `SET` and `CALL` events. The first lines of the report cover the decryption of the obfuscated string, which is finally revealed in lines 232–233. Starting with the call to `eval`, this string is evaluated by the interpreter and results in the construction of a NOP sled with 1024 bytes in line 246.

```
SET  global.a TO ""
SET  global.b TO "{@xqhvfdsh+%(x<3<3%,>zkloh
                  +{1ohqjwk?4333,{.@{>"
SET  global.i TO "0"
CALL charCodeAt
SET  global.c TO "120"
CALL fromCharCode
SET  global.a TO "x"
...
SET  global.a TO "x=unescape("%u9090");
                   while(x.length<1024)x+=x;"
SET  global.i TO "46"
CALL eval
CALL unescape
SET  global.x TO "<90>"
SET  global.x TO "<90><90>"
...
SET  global.x TO "<90><90> ... 1024 bytes ...  <90>"
```

Figure 4: Example of dynamic analysis report.

2.2 Feature Extraction

The static and dynamic analysis of JavaScript code provide a wealth of information for identifying malicious activity. In contrast to related methods, however, we do not manually define features indicative for this activity, but instead apply a generic approach for feature extraction. This approach is independent of particular attack types and enables us to model the analysis geometrically in a vector space.

The reports generated by the static and dynamic analysis can be interpreted as sequences of words. While the lexical analysis naturally returns such a sequence, the behavior reports obtained from ADSandbox can be partitioned into words using whitespace characters. For both analysis types, we move a fixed-length window over the words of each sequence and extract subsequences of q words at each position, so-called *q-grams*. The following example shows the extraction of q-grams with $q = 3$ for a short snippet of the static analysis,

$$\texttt{ID = ID + NUM} \\ \longrightarrow \big\{(\texttt{ID = ID}), (\texttt{= ID +}), (\texttt{ID + NUM})\big\}.$$

As a result of this extraction, each analysis report is represented by a set of q-grams, which reflect short patterns of its content. To establish a map from these patterns to a vector space we associate each q-gram with one dimension in the vector space. Formally, this vector space is defined using the set S of all possible q-grams, where the mapping for an analysis report x is given by $\phi : x \to \big(\phi_s(x)\big)_{s\in S}$ with

$$\phi_s(x) = \begin{cases} 1 & \text{if } x \text{ contains the } q\text{-gram } s, \\ 0 & \text{otherwise.} \end{cases}$$

The function ϕ maps a report x to the vector space $\mathbb{R}^{|S|}$, such that all dimensions associated with q-grams contained in x are set to one and all others are zero. Note that although the induced vector space is high-dimensional, its sparse structure still allows for very efficient computations [see 23].

2.3 Learning-based Detection

The geometric representation of the analysis reports enables us to apply techniques from the domain of machine learning for generating detection models, instead of manually crafting detection rules. In particular, Cujo exploits the power of *Support Vector Machines* (SVM) [26] that are known for very effective and robust learning in various applications.

Given vectors of two classes as training data, an SVM learns a hyperplane in the vector space that separates the two classes with maximum margin. In the case of Cujo, these classes correspond to analysis reports of benign (−) and malicious (+) JavaScript code, as depicted in Figure 5.

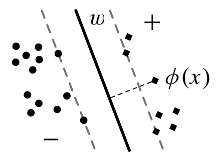

Figure 5: Hyperplane with maximal margin.

Formally, an SVM learns a detection model by determining a vector w and bias b, specifying the direction and offset of the maximum-margin hyperplane. The detection function f of an SVM is given by

$$f(x) = \langle \phi(x), w \rangle + b = \sum_{s\in S} \phi_s(x) \cdot w_s + b.$$

and returns the orientation of $\phi(x)$ with respect to the hyperplane. That is, $f(x) > 0$ indicates malicious activity in the analysis report x and $f(x) \leq 0$ corresponds to benign data. In practice, this computation can be carried out very efficiently with a median run-time below 0.2 ms per report [24]. For large-scale learning of detection models, Cujo makes use of LibLinear [12], a fast SVM library that enables to train detection models from 100,000 web pages in 120 seconds for dynamic analysis and in 50 seconds for static analysis.

The learning-based detection completes the design of Cujo. As illustrated in Figure 1, Cujo uses two independent processing chains for static and dynamic code analysis, where an alert is reported if one of the detection models indicates an attack. This combined detection renders evasion of Cujo difficult, as it requires the attacker to cloak his attacks from both, static and dynamic analysis.

3 Training Architecture

So far we have seen how a learning-based detector for malicious JavaScript code can be constructed. To ensure a long-term protection, however, a detector needs to be regularly updated and trained on benign and malicious JavaScript code. Previous work has largely ignored this need for retraining and evaluated detectors "in vitro" using manually collected samples [e.g. 5, 13, 24]. In practice, retraining needs to be repeated regularly "in vivo" and automatic means for collecting and sanitizing training data are crucial for operating a learning-based detector. As a remedy, we introduce an architecture for automatically retraining Cujo and other learning-based detectors in this article.

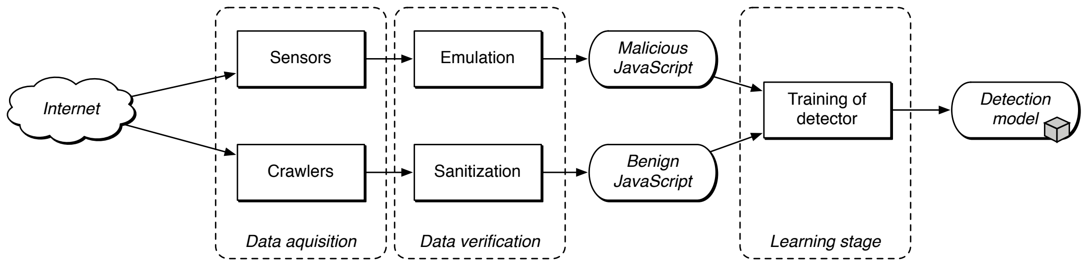

Figure 6: An architecture for retraining Cujo.

Figure 6 shows a schematic overview of our training architecture. The architecture contains three layers of components, which account for the acquisition of data, its verification, and the learning process. In particular, the verification of collected data is a critical issue for automatic retraining. First, malicious code hidden in benign web pages needs to be identified and removed. Second, benign code accidentally added as attacks needs to be filtered from the malicious data.

3.1 Data Acquisition

The first layer of the architecture is designed for automatically collecting potentially benign and malicious JavaScript code. This task is accomplished using different realizations of so-called *crawlers* and *sensors*.

Crawlers are used to retrieve benign web pages. For this purpose, crawlers regularly query rankings and listings of popular web pages and download large sets of corresponding JavaScript code. In our current implementation we randomly sample from the Alexa ranking[1] which lists the top 1 million web pages according to daily visitors and page views. Popular web pages are not guaranteed to be attack-free—in fact they are frequent targets of web-based attacks. However, by visiting a large sample of popular web pages, say 20.000 per day, the fraction of contained attacks can be constrained, such that the majority of downloaded code is benign.

Sensors take on the opposite task and retrieve malicious JavaScript code from web pages. To this end, sensors either actively seek for malicious activity using low-interaction honeyclients [3, 14, 19] or passively acquire attacks using monitoring techniques, such as spamtraps. Moreover, databases of malicious URLs maintained by community projects, such as Harmur [17], also provide a valuable source for malicious code. Similarly to crawlers, the code downloaded by sensors is not guaranteed to be malicious and thus the verification of all collected data is a prerequisite for automatic retraining.

3.2 Data Verification

In the second layer of the architecture the collected data is verified using different techniques. In particular, we build on recent work from the areas of honeypots and intrusion detection systems to devise an automatic verification of benign and malicious JavaScript code.

To determine whether a collected code sample is malicious we employ different offline analysis techniques, which are accurate in identifying malicious activity but too inefficient for directly protecting web clients. Common implementations for offline analysis include honeypots that operate in a virtual machine, such as Capture-HPC [27] and Shelia [25], and sophisticated sandboxes, such as Jsand [5] and Rozzle [16]. By combining diverse analysis techniques, we obtain a more accurate prediction of whether a code sample is malicious. Although we cannot computationally verify maliciousness in this setting, we get a clear indication for unwanted JavaScript code that should be kept away from web clients.

While there are several techniques for identifying malicious patterns in code, there exist no tools for testing whether a code snippet is benign—simply due to the sheer diversity and complexity of JavaScript code in the Internet. As a remedy, we implement the concept of *sanitization*, originally proposed for network anomaly detection [6]. We proceed by partitioning the benign data into several small blocks and train a detector on each of these blocks. The learned detectors are then applied to the remaining blocks, such that each code sample is judged by several detectors. Using a majority voting, we can identify code that is not benign with high probability.

This sanitization succeeds in removing unknown attacks from the collected data for two reasons: First, the majority of acquired data is benign and most detectors still make correct predictions. Second, the unknown attacks are not uniformly distributed over all blocks and thus each detector is only influenced by a subset of malicious code [see 6].

3.3 Learning Stage

Once a data set of malicious and benign code is available, a learning-based detector can be automatically trained using the data collected in a fixed time period, such as a week. Similarly, parameters of the detector, e.g. for feature extraction, can be tuned using cross-validation on the collected data [2]. While our architecture has been primarily devised for Cujo, its design is agnostic to the detector and, for example, also suitable for training Zozzle and IceShield.

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

4 Empirical Evaluation

We proceed to study the ability of Cujo to detect malicious JavaScript code in practice. So we compare Cujo with related approaches and analyze how the proposed architecture for training enables learning in the presence of attacks. A comprehensive evaluation of Cujo, including a detailed study of its run-time behavior, is provided by Rieck et al. [24].

For the evaluation, we consider a data set of 200,000 benign web pages corresponding to the top sites listed by Alexa, as detailed in Section 3.1. To ensure that our initial training data is not already contaminated with malicious code, we scan the data for common attack strings and use the GoogleSafeBrowsing service to sort out potential attacks. As malicious data set, we use a collection of 609 real JavaScript attacks that has been gathered using the Wepawet services over a period of two years [see 5, 24].

4.1 Detection Performance

In our first experiment we analyze the ability of Cujo to detect unseen JavaScript attacks. Hence, we split our benign and malicious data set into two parts, a training partition (75%) and a testing partition (25%). Cujo is trained and calibrated on the training partition, whereas the detection performance is measured on the testing partition. To get a statistically sound estimate of the detection performance this procedure is repeated 10 times and average results are reported.

Cujo detector	**TP rate**	**FP rate**
static only	90.2%	0.001%
dynamic only	86.0%	0.001%
static & dynamic	94.4%	0.002%

Table 1: Detection performance of Cujo.

Table 1 presents the detection performance of Cujo in terms of true-positive rate (ratio of detected attacks) and false-positive rate (ratio of benign web pages flagged as malicious). When looking at the static and dynamic analysis of Cujo alone, we note that both already identify the majority of attacks. By combining both analyses, the performance of Cujo is further boosted, reaching a detection of 94%. Moreover, Cujo attains a false-positive rate of 0.002%, corresponding to just 2 false alarms in 100,000 visited web pages.

Anti-virus scanners	**TP rate**	**FP rate**
ClamAV	35.0%	0.000%
Avira AntiVir	70.0%	0.087%

Table 2: Detection performance of two anti-virus scanners.

For comparison, the detection performance of two regular anti-virus scanners is shown in Table 2. Both scanners fail to accurately identify the JavaScript attacks in our experiment, although they have been equipped with the latest signatures.

Learning-based detectors	**TP rate**	**FP rate**
JSand [5]	99.8%	0.013%
Zozzle [7]	90.8%	0.000%
Prophiler [4]	99.2%	9.800%
IceShield [13]	98.0%	2.179%

Table 3: Detection performance of other learning-based detectors. The results have been taken from the respective publications.

Results for the detection performance of other learning-based detectors are shown in Table 3. These results have been obtained on different data sets and thus only larger differences can be considered for comparison. The best results are achieved by JSand, an anomaly detector intergrated in the Wepawet service. However, JSand is designed for offline analysis and not capable of detecting attacks in real-time. From the detectors suitable for on-line application, Cujo and Zozzle perform on par and attain the best performance, where Zozzle identifies slightly less attacks with a lower false-positive rate. In comparison, Cujo is among the best approaches for identifying and stopping malicious JavaScript code in the Internet.

4.2 Robustness

For a reliable day-to-day operation, learning-based detectors need to be regularly retrained. In Section 3 we have introduced an architecture capable of automatically collecting and verifying training data. We now empirically evaluate the efficacy of this architecture and study how the presence of unknown attacks impacts the detection performance of Cujo.

For this experiment, we add 50% of the JavaScript attacks to the training data and label them as benign code, thereby generating a contaminated data set. This setup resembles almost a worst-case scenario, as some attacks from the testing partition are now present in the training data as benign code. For the evaluation, we either train Cujo directly on the contaminated data or apply the sanitization procedures outlined in Section 3.2 before training. We then proceed as in the previous experiment.

Cujo detector	**TP rate**	**FP rate**
trained on clean data	94.4%	0.002%
trained on contaminated data	87.9%	0.001%
trained on sanitized data	93.2%	0.003%

Table 4: Impact of unknown attacks on detection performance.

Table 4 shows results of this experiment. The unknown attacks in the training data impact the true-positive rate indicating that Cujo is effected by the contaminated data. With the sanitized training data, however, Cujo performs almost identical to the previous experiment, although half of the attacks are labeled as benign data before sanitization. This result confirms that the proposed architecture enables accurate training of detectors, even in presence of unknown attacks.

5 Related Work

Since the first discovery of web-based attacks, the detection and mitigation of this threat has been a vivid area of security research. Several approaches have been devised for observing, analyzing, and detecting JavaScript attacks, for example, using high-interaction honeypots [21, 27, 29] and low-interaction honeypots [3, 14, 19]. Similarly, different systems have been proposed for offline analysis of JavaScript code [4, 5, 15]. While all these approaches are effective in detecting malicious code, they require considerable analysis time and thus are inapplicable for protecting users at run-time.

Concurrently to these offline approaches, other work has studied the detection and prevention of certain attack types, such as heap-spraying attacks [11, 22] and drive-by downloads [18]. This work rests on identifying symptoms of particular attacks, for example, the presence of shellcode in JavaScript strings. Although these approaches mitigate the threat of web-based attacks to some extent, they are limited to specific attack types and unable to cope with the evolving domain of malicious JavaScript code.

As a consequence, recent work has studied combining JavaScript analysis with techniques from machine learning for deriving automatic defenses. Most notably are the learning-based detection systems Cujo [24], Zozzle [7], and IceShield [13]. In this article, we have presented Cujo which shares many similarities with the other two systems. However, Cujo differs in two key aspects from related work: First, it combines static and dynamic analysis of JavaScript code, whereas, for example, Zozzle builds on static inspection of code and IceShield relies on run-time monitoring only. Second, related detectors operate from within a web browser and thereby gain efficient access to JavaScript code. Cujo is embedded in a web proxy, which requires more effort for processing code, but in turn enables transparently protecting multiple and heterogeneous web clients.

6 Conclusions

In this article we have presented a learning-based detector for malicious JavaScript code, along with a corresponding training architecture. Our detector, Cujo, combines a lightweight static and dynamic analysis for obtaining a holistic view on JavaScript code. This view enables us to identify malicious patterns in the code as well as the behavior of web pages, such that malicious content can blocked prior to its delivery to the user. To operate the detector in practice and to update the underlying detection models we propose a training architecture that enables to automatically collect malicious and benign code for training. The architecture builds on recent concepts of honeypots and intrusion detection systems for verifying and sanitizing the collected code automatically.

In an empirical evaluation with 200,000 web pages and several hundred JavaScript attacks, Cujo significantly outperforms regular anti-virus tools and enables a detection of 94% of the attacks with only 2 false alarms in 100,000 visited web pages. The capabilities of the proposed architecture are demonstrated when attacks are inter-mixed with benign web pages. In this setting Cujo still identifies over 93% of the attacks, even if most of these are mixed with benign web pages during the training process.

Cujo and other learning-based detectors provide a valuable tool for detecting and blocking malicious JavaScript code. Together with other defenses, such as honeyclients and offline analysis, they enable containing and ultimately stopping the proliferation of web-based attacks. In line with this goal, we are currently deploying Cujo and the proposed architecture in different settings to study its efficacy in the wild.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[2] S. Arlot, A. Celisse, and P. Painleve. A survey of cross-validation procedures for model selection. *Statistics Surveys*, 4:40–79, 2010.

[3] 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.

[4] 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.

[5] 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.

[6] G. Cretu, A. Stavrou, M. Locasto, S. Stolfo, and A. Keromytis. Casting out demons: Sanitizing training data for anomaly sensors. In *Proc. of IEEE Symposium on Security and Privacy*, pages 81–95, 2008.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. Technical Report MSR-TR-2011-94, Microsoft Research, Aug. 2011.

[17] 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.

[18] 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.

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

[20] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. In *Proc. of USENIX Workshop on Hot Topics in Understanding Botnets (HotBots)*, 2007.

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

[22] 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.

[23] K. Rieck and P. Laskov. Linear-time computation of similarity measures for sequential data. *Journal of Machine Learning Research*, 9(Jan):23–48, Jan. 2008.

[24] 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.

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

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

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

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

[29] 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.