Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

CUJO: Efficient Detection and Prevention of Drive-by-Download Attacks

Konrad Rieck, Tammo Krueger, and Andreas Dewald

Bericht-Nr. 2010-10
ISSN 1436-9915

Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks

Konrad Rieck[1], Tammo Krueger[1], and Andreas Dewald[2]

[1] Berlin Institute of Technology, Germany
[2] University of Mannheim, Germany

Abstract

The JavaScript language is a core component of active and dynamic web content in the Internet today. Besides its great success in enhancing web applications, however, JavaScript provides the basis for drive-by downloads—attacks exploiting vulnerabilities in web browsers and their extensions for unnoticeably downloading malicious software. Due to the diversity and frequent use of obfuscation in these JavaScript attacks, static code inspection proves ineffective in practice. While dynamic analysis and honeypots provide means to identify drive-by-download attacks, current approaches induce a significant overhead which renders immediate prevention of attacks intractable.

In this paper, we present Cujo, a system for automatic detection and prevention of drive-by-download attacks. Embedded in a web proxy, Cujo transparently inspects web pages and blocks delivery of malicious JavaScript code. Static and dynamic code features are extracted on-the-fly and analysed for malicious patterns using efficient techniques of machine learning. We demonstrate the efficacy of Cujo in different experiments, where it detects 95% of the drive-by downloads with few false alarms and a median run-time of 500 ms per web page—a quality that, to the best of our knowledge, has not been attained in previous work on detection of drive-by-download attacks.

1 Introduction

The JavaScript language is a ubiquitous tool for providing active and dynamic content in the Internet. The vast majority of web sites, including large social networks, such as Facebook and Twitter, makes heavy use of JavaScript for enhancing the appearance and functionality of their services. In contrast to server-based scripting languages, JavaScript code is executed in the web browser of the client and thus provides means for directly interacting with the user and the browser environment. Although the execution of JavaScript code at the client is restricted by several security policies, the interaction with the browser and its extensions alone gives rise to a severe security threat.

JavaScript is increasingly used as basis for so-called *drive-by downloads*, attacks exploiting vulnerabilities in web browsers and their extensions for unnoticeably downloading malicious software [see 15, 16]. These attacks take advantage of the tight integration of JavaScript with the browser environment to exploit different types of vulnerabilities and eventually assume control of the web client. Due to the complexity of browsers and their extensions, there exist numerous of these vulnerabilities, ranging from insecure interfaces of third-party extensions to buffer overflows and memory corruptions [5, 7, 12]. Four of the top five most attacked vulnerabilities observed by Symantec in 2009 have been such client-side vulnerabilities and involved in drive-by-download attacks [2].

As a consequence, detection of drive-by downloads has gained a focus in security research. Two classes of defense measures have been proposed to counteract this threat: First, several security vendors have equipped their products with rules and heuristics for identifying malicious code directly at the client. This static code inspection, however, is largely obstructed by the frequent use of obfuscation in drive-by downloads. A second strain of research has thus studied detection of drive-by downloads using dynamic code analysis, for example using code emulation [8, 17], sandboxing [4, 6, 16] and client honeypots [14, 16, 19]. Although effective in detecting attacks, these approaches suffer from either of two shortcomings: Some are limited to specific attack types, such as heap spraying [e.g., 8, 17], whereas the more general [e.g., 4, 14] induce an overhead prohibitive for preventing attacks at the client.

As a remedy, we present Cujo[1], a system for detection and prevention of drive-by-download attacks, which combines advantages of static and dynamic analysis concepts. Embedded in a web proxy, Cujo transparently inspects web pages and blocks delivery of malicious JavaScript code to the client. The analysis and detection methodology implemented in this system rests on the following contributions of this paper:

- *Lightweight JavaScript analysis.* We devise efficient techniques for static and dynamic analysis of JavaScript code contained in web pages, which provide expressive analysis results with very small run-time overhead.
- *Generic feature extraction.* For generic detection of attacks, we introduce a mapping from analysis reports to a vector space that is spanned by short analysis patterns and independent of particular attack characteristics.
- *Learning-based detection.* We apply techniques of machine learning for generating detection models for static and dynamic analysis, which spares us from manually crafting and updating detection rules as in current security products.

An empirical evaluation with 200,000 web pages and 600 real drive-by-download attacks demonstrates the efficacy of this approach: Cujo detects 95% of the attacks with a false-positive rate of 0.004%, corresponding to 4 false alarms in 100,000 visited web sites, and thus is almost on par with offline analysis systems, such as Jsand [4]. In terms of run-time, however, Cujo significantly surpasses these systems. With caching enabled, Cujo provides a median run-time of 500 ms per web page, including downloading of web page content and full analysis of JavaScript code. To the best of our knowledge, Cujo is the first system capable to effectively and efficiently block drive-by-download attacks in practice.

The rest of this paper is organized as follows: Cujo and its detection methodology are introduced in Section 2 including JavaScript analysis, feature extraction and learning-based

[1]Cujo = "Classification of Unknown JavaScript Objects"

detection. Experiments and comparisons to related techniques are presented in Section 3. Related work is discussed in Section 4 and Section 5 concludes.

2 Methodology

Drive-by-download attacks can take almost arbitrary structure and form, depending on the exploited vulnerabilities as well as the use of obfuscation. Efficient analysis and detection of these attacks resembles a challenging problem, which requires careful balancing of detection and run-time performance. We address this problem by applying lightweight static and dynamic code analysis, thereby providing two complementary views on JavaScript code. To avoid manually crafting detection rules for each of these views, we employ techniques of machine learning, 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.

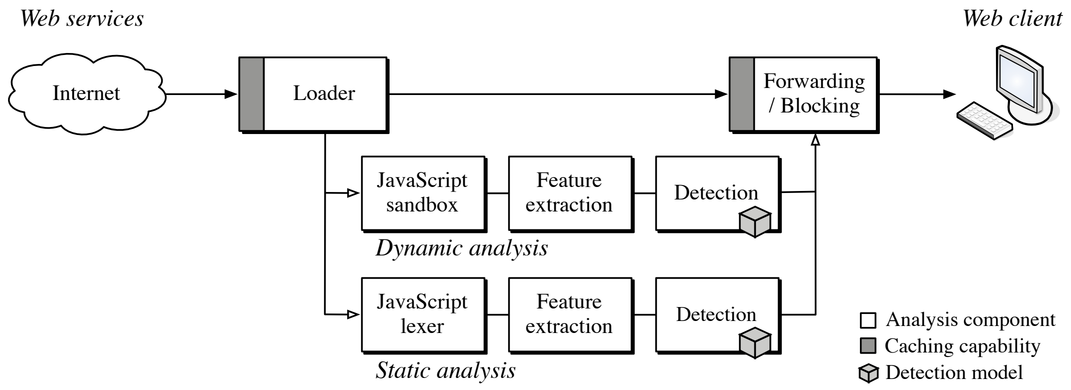


Figure 1: Schematic depiction of Cujo. Incoming web data is processed using static and dynamic analysis of JavaScript code. Depending on individual analysis results, the data is either forwarded to the Web client or blocked.

Cujo is embedded in a web proxy and transparently inspects communication between a web client and a web service. Prior to delivery of web page data from the service to the client, Cujo performs a series of analysis steps and depending on their results blocks pages likely containing malicious JavaScript code. To improve processing performance, two analysis caches are employed: First, all incoming web data is cached to reduce loading times and, second, analysis results are cached, if all embedded and external code associated with a web page has not changed within a limited period of time.

2.1 JavaScript Analysis

As first analysis step, we aim at efficiently getting a comprehensive view on JavaScript code. To this end, we inspect all HTML and XML documents passing our system for occurrences of JavaScript. For each requested document, we extract all code blocks embedded using the HTML tag `script` and contained in HTML event handlers, such as `onload` and `onmouseover`.

Moreover, we recursively pre-load all external code referenced in the document, including scripts, frames and iframes, to obtain the complete code base of the web page. All code blocks of a requested document are then merged for further static and dynamic analysis.

As an example running the following sections, we consider the JavaScript code shown in Figure 2(a). The code is obfuscated using a simple substitution cipher and contains a routine for constructing a NOP sled, an array of NOP instructions common in most memory corruption attacks. Analysis reports for the static and dynamic analysis of this code snippet are shown in Figure 2(b) and 2(c), respectively.

```
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);
```

(a) Obfuscated JavaScript code

```
ID = STR.00 ;
ID = STR.02 +
     STR.02 ;
FOR ( ID = NUM ; ID < ID . ID ; ID ++ ) {
    ID = ID . ID ( ID ) - NUM ;
    ID + = ID . ID ( ID ) ;
}
EVAL ( ID ) ;
```

(b) Static analysis: Report of lexical tokens

```
  1 SET  global.a TO ""
  2 SET  global.b TO "{@xqhvfdsh+%(x<3<3%,>zkloh+{1ohqjwk?4333,{.@{>"
  3 SET  global.i TO "0"
  4 CALL charCodeAt
  5 SET  global.c TO "120"
  6 CALL fromCharCode
  7 SET  global.a TO "x"
    ...
232 SET  global.a TO "x=unescape("%u9090");while(x.length<1000)x+=x;"
233 SET  global.i TO "46"
234 CALL eval
235 CALL unescape
236 SET  global.x TO "<90><90>"
    ...
246 SET  global.x TO "<90><90> ... 1024 bytes ... <90><90>"
```

(c) Dynamic analysis: Behavior report

Figure 2: Example of static and dynamic JavaScript analysis: (a) Obfuscated code snippet of a NOP sled generation, (b) lexical tokens extracted using static analysis, (c) a behavior report generated using dynamic analysis. The deobfuscated code is visible in line 232.

Static analysis. Our static analysis relies on basic principles of compiler design [3]: Before the source code of a program can be interpreted or compiled, it needs to be decomposed into lexical tokens, which are then fed to the actual parser. The static analysis component in Cujo takes advantage of this process and efficiently extracts lexical tokens from the JavaScript code of a web page using a customized Yacc grammar.

This lexical analysis closely follows the language specification of JavaScript [1], where source code is sequentially decomposed into keywords, punctuators, identifiers and literals. As the actual names of identifiers do not contribute to the structure of code, we replace them by the generic token `ID`. Similarly, we encode numerical literals by `NUM` and string literals by `STR`. An example of this basic decomposition is illustrated in the following

```
x = foo(y) + "bar";    ⟶    ID = ID ( ID ) + STR ;
```

where keywords and punctuators are represented by individual tokens, while identifiers and strings are subsumed by the generic tokens `ID` and `STR`, respectively.

To further strengthen our static analysis for detection of drive-by-download attacks, we make two refinements to the lexical analysis. First, we additionally encode the length of string literals as decimal logarithm. That is, `STR.01` refers to a string with up to 10^1 characters, `STR.02` to a string with up to 10^2 characters and so on. Second, we add `EVAL` as a new keyword to the lexical analysis. Both refinements target common constructs of drive-by-download attacks, which frequently involve string operations and calls to the `eval()` function.

Although obfuscation techniques may hide code from this static analysis, several programming constructs and structures can be distinguished in terms of lexical tokens. As an example, Figure 2(b) shows an analysis report of lexical tokens. While the code for generating a NOP sled is hidden in the encrypted string (line 2–3), several patterns indicative for obfuscation, such as the decryption loop (line 4–6) and the call to `EVAL` (line 8), are accessible to means of detection techniques

Dynamic analysis. For dynamic analysis, we adopt an enhanced version of ADSandbox, a lightweight JavaScript sandbox developed by Dewald et al. [6]. The sandbox takes the code associated with a web page and executes it within the JavaScript interpreter SpiderMonkey[2]. The interpreter operates in a virtual browser environment and reports all operations changing the state of this environment. Additionally, we invoke all event handlers of the code to trigger functionality dependent on external events. As result of this analysis, the sandbox provides a report containing all monitored operations, as shown in Figure 2(c).

To emphasize behavior related to drive-by-download attacks, we extend the dynamic code analysis with *abstract operations*, which represent patterns of common attack activity. These abstract operations are encoded as regular expressions and matched on-the-fly during the monitoring of a JavaScript code. Currently, Cujo supports two of these operations: First, we indicate typical behavior of heap spraying attacks, such as excessive allocation of memory chunks by appending the operation `HEAP SPRAYING` and, second, we mark the use of browser functions inducing a re-evaluation of strings by the interpreter using the operation `PSEUDO-EVAL`. While both abstract operations are indicative for particular attacks, they are not sufficient for detection alone and a full inspection of behavior reports is required.

Although this lightweight analysis provides only a coarse view on the behavior of JavaScript code in comparison to offline analysis [e.g., 4, 14, 19], it enables accurate detection of drive-by downloads with a median run-time of less than 400 ms per web page, as demonstrated in

[2]SpiderMonkey (JavaScript-C) Engine – `http://www.mozilla.org/js/SpiderMonkey`

Section 3.3. As an example, Figure 2(c) shows a behavior report for the code snippet given in Figure 2(a). The first lines of the report cover the decryption of the obfuscated string, which is finally revealed in line 232. 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.

2.2 Feature Extraction

In the second analysis step of Cujo, we extract features from the analysis reports of static and dynamic analysis, suitable for application of detection methods. In contrast to previous work, we propose a generic feature extraction, which is independent of particular attack characteristics and allows to jointly process reports of static and dynamic analysis.

Q-grams features. Our feature extraction builds on the concept of q-grams, which has been widely studied in the field of intrusion detection [e.g., 11, 20]. To unify the representation of static and dynamic analysis, we first partition each report into a sequence of words using white-space characters. We then move a fixed-length window over each report and extract subsequences of q words at each position, so-called q-grams. The following example shows the extraction of q-grams for two snippets of analysis reports with $q = 3$,

$$\begin{aligned} \texttt{ID = ID + NUM} \quad &\longrightarrow \quad \big\{ (\texttt{ID = ID}), (\texttt{= ID +}), (\texttt{ID + NUM}) \big\} \\ \texttt{SET global.a to "x"} \quad &\longrightarrow \quad \big\{ (\texttt{SET global.a to}), (\texttt{global.a to "x"}) \big\}. \end{aligned}$$

As a result, each report is represented by a set of q-grams, which reflect short analysis patterns and provide the basis for mapping analysis reports to a vector space.

Intuitively, we are interested in constructing a vector space, where analysis reports sharing several q-grams lie close to each other, while reports with dissimilar content are separated by large distances. To establish such a mapping, we associate each q-gram with one particular dimension in the vector space. Formally, this vector space is defined using the set S of all possible q-grams, where a corresponding mapping function for a report x is given by

$$\phi : x \longrightarrow \big(\phi_s(x)\big)_{s \in S} \quad \text{with} \quad \phi_s(x) = \begin{cases} 1 & \text{if } x \text{ contains the } q\text{-gram } s, \\ 0 & \text{otherwise.} \end{cases}$$

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 other dimensions are zero. To avoid an implicit bias on the length of reports, we normalize $\phi(x)$ to one, that is, we set $||\phi(x)|| = 1$. This procedure ensures that patterns contained in a report x contribute to the vector $\phi(x)$ only according to their relative length in x.

Efficient computation. At the first glance, this mapping seems intractable and inappropriate for efficient analysis: the set S covers all possible q-grams of words and thus induces a vector space of arbitrarily large dimension. However, the number of q-grams contained in a report is linear in its length. That is, a report x containing m words comprises at most $(m - q)$ different q-grams. Consequently, only $(m-q)$ dimensions are non-zero in the vector $\phi(x)$—irrespective of the dimension of the vector space. It thus suffices to only store the q-grams contained in each report x for sparsely representing the vector $\phi(x)$, for example, using efficient data structures such as Tries [10] or Bloom filters [20]. As demonstrated in Section 3.3, this sparse representation of feature vectors provides the basis for very efficient feature extraction with median run-times below 1 ms per analysis report.

2.3 Learning-based Detection

As final analysis step of Cujo, we present a learning-based detection of drive-by-download attacks, which builds on the vectorial representation of analysis reports. The application of machine learning spares us from manually constructing and updating detection rules for static and dynamic code analysis, and limits the delay to detection of novel drive-by downloads.

Support Vector Machines. For automatically generating detection models from the reports of attacks and benign code, we apply the technique of *Support Vector Machines* (SVM) [see 18]. Given vectors of two classes as training data, an SVM determines an hyperplane that separates both classes with maximal margin. In our setting, one of these classes is associated with analysis reports of drive-by downloads, whereas the other class corresponds to reports of benign web pages. An unknown report is classified by mapping it to the vector space and checking if it falls on the malicious (+) or benign (-) side of the hyperplane.

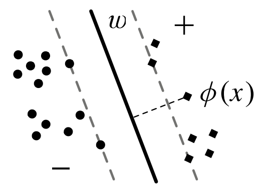


Figure 3: Hyperplane with maximal margin.

Formally, the detection model of an SVM corresponds to a vector w and bias b, specifying the direction and offset of the hyperplane. The corresponding detection function f 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 report x and $f(x) \leq 0$ corresponds to benign data.

In contrast to many other learning techniques, SVMs possess the ability to compensate a certain amount of noise in the labels of the training data—a crucial property for practical application of Cujo. This ability renders the learning process robust to a minor amount of unknown attacks in the benign portion of the training data and enables generating accurate detection models, even if some of the web pages labeled as benign data contain drive-by-download attacks. Theory and further details on this ability are provided in [18].

Efficient computation. For efficiently computing f, we again exploit the sparse representation of vectors induced by ϕ. Given a report x, we know that only q-grams contained in x have non-zero entries in $\phi(x)$, that is, all other dimensions in $\phi(x)$ are zero and do not contribute to the computation of $f(x)$. Hence, we can simplify the detection function f as follows

$$f(x) = \sum_{s \in S} \phi_s(x) \cdot w_s + b = \sum_{s \text{ in } x} \phi_s(x) \cdot w_s + b,$$

where we determine $f(x)$ by simply looking up the values w_s for each q-gram contained in x. As a consequence, the classification of a report can be carried out with linear time complexity and a median run-time below 0.2 ms per report (cf. Section 3.3). For learning the detection model of the SVM we employ LibLinear [9], a fast SVM library which enables us to train detection models from 100,000 reports in 120 seconds for dynamic analysis and in 50 seconds for static analysis.

Explanation. In practice, a detection systems must not only flag malicious events but also provide insights into the detection process, such that attack patterns and exploited vulnerabilities can be inspected during operation. Fortunately, we can adapt the detection function for explaining the decision process of the SVM. During computation of f, we additionally store the individual contribution $\phi_s(x) \cdot w_s$ of each q-gram to the final detection score $f(x)$. If an explanation is requested, we output the q-grams with largest contribution and thereby present those analysis patterns that shifted the analysis report x to the positive side of the hyperplane. We illustrate this concept in Section 3.2, where we present explanations for detections of drive-by-download attacks using dynamic analysis reports.

The learning-based detection completes the design of our system 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 a drive-by download.

This combined detection renders evasion of our system difficult, as it requires the attacker to cloak his attacks from both, static and dynamic analysis. While static analysis alone can be thwarted through massive obfuscation, the hidden code needs to be decrypted during run-time which in turn can be tracked by dynamic analysis. Similarly, if fewer obfuscation is used and the attacker tries to spoil the sandbox emulation, patterns of the respective code might be visible to static analysis. Although this argumentation does not rule out evasion in general, it clearly shows the effort necessary for evading our system.

3 Evaluation

After presenting the detection methodology of Cujo, we turn to an empirical evaluation of its performance. In particular, we conduct experiments to study the detection and run-time performance in detail. Before presenting these experiments, we introduce our data sets of drive-by-download attacks and benign web pages.

Data sets. We consider two data sets containing URLs of benign web pages, *Alexa-200k* and *Surfing*, which are listed in Table 1(a). The *Alexa-200k* data set corresponds to the 200,000 most visited web pages in Internet as listed by Alexa[3] and thus covers a wide range of JavaScript code, including several search engines, social networks and on-line shops. The *Surfing* data set comprises 20,283 URLs of web pages visited during usual web surfing at our institute. The data has been recorded over a period of 10 days and contains individual sessions of five users. Both data sets have been sanitized by scanning the web pages for drive-by downloads using common

[3] Alexa Top Sites – `http://www.alexa.com/topsites`

attack strings and the GoogleSafeBrowsing service. While very few unknown attacks might still be present in the data, we rely on the ability of the SVM learning algorithm to compensate this inconsistency effectively.

(a) Benign data sets

Data set	# URLs
Alexa-200k	200,000
Surfing	20,283

(b) Attack data sets

Data set	# attacks	Data set	# attacks
Spam Trap	256	Wepawet-new	46
SQL Injection	22	Obfuscated	84
Malware Forum	201		

Table 1: Description of benign and attack data sets. The attack data sets have been mainly taken from Cova et al. [4] and adapted to contain valid HTML documents.

The attack data sets are listed in Table 1(b) and have been mainly taken from Cova et al. [4]. In total, the attack data sets comprise 609 HTML documents containing several types of drive-by-download attacks collected over a period of two years. The attacks are organized according to their origin: the *Spam Trap* set comprises attacks extracted from URLs in spam messages, the *SQL Injection* set contains drive-by downloads injected into benign web sites, the *Malware Forum* set covers attacks published in Internet forums, and the *Wepawet-new* set contains malicious JavaScript code submitted to the Wepawet service[4]. A detailed description of these classes is provided in [4]. Moreover, we provide the *Obfuscated* set which contains 28 attacks from the other sets additionally obfuscated using a popular JavaScript packer[5].

3.1 Detection Performance

In our first experiment, we study the detection performance of Cujo in terms of true-positive rate (ratio of detected attacks) and false-positive rate (ratio of misclassified benign web pages). As the learning-based detection implemented in Cujo requires a set of known attacks and benign data for constructing detection models, we conduct the following experimental procedure: We randomly split all data sets into a *known* and an *unknown* partition. The detection models and respective parameters, such as the best length of q-grams, are determined on the known partition, whereas the unknown partition is only used for measuring the final detection performance. We repeat this procedure 10 times and average results. The partitioning ensures that reported results only refer to attacks unknown during the learning phase of Cujo.

For comparing the performance of Cujo with state-of-the-art methods, we also consider static detection methods, namely the anti-virus scanner ClamAv[6] and the web proxy of the security suite AntiVir[7]. As ClamAv does not provide any proxy capabilities, we manually feed the downloaded web pages and respective JavaScript code to the scanner. Moreover, we add results presented by Cova et al. [4] for the offline analysis system Jsand to our evaluation.

[4] Wepawet Service – `http://wepawet.cs.ucsb.edu`

[5] JavaScript Packer by Dean Edward – `http://dean.edwards.name/packer`

[6] Clam AntiVirus – `http://www.clamav.net/`

[7] Avira AntiVir Premium – `http://www.avira.com/`

Attack data sets		Cujo		ClamAv	AntiVir	Jsand
	static	dynamic	combined			
Spam Trap	96.5%	98.8%	99.4%	41.0%	58.2%	99.7%
SQL Injection	91.4%	86.7%	99.2%	18.2%	95.5%	100.0%
Malware Forum	77.1%	79.4%	86.7%	45.3%	83.1%	99.6%
Wepawet-new	87.0%	86.1%	95.6%	19.6%	93.5%	—
Wepawet-old	—	—	—	—	—	100.0%
Obfuscated	100.0%	93.3%	100.0%	4.8%	54.8%	—
Total	89.7%	90.2%	95.0%	35.0%	70.0%	99.8%

Table 2: Comparison of true-positive rates on attack data sets. Results for Cujo have been averaged over 10 independent runs. Results for Jsand have been taken from Cova et al. [4]. The Wepawet-new data set is a recent version of Wepawet-old.

True-positive rates. Table 2 lists the detection performance in terms of true-positive rates for Cujo and the other detection methods. The static and dynamic code analysis of Cujo alone attain a true-positive rate of 89.7% and 90.2%, respectively. The combination of both, however, allows to identify 95% of the attacks, demonstrating the advantage of two complementary views on JavaScript code. A better performance is achieved by Jsand which is able to almost perfectly detect all attacks. However, Jsand generally operates offline and spends considerably more time for analysis of JavaScript code. The anti-virus tools, ClamAv and AntiVir, achieve lower detection rates of 35% and 70%, respectively, although both have been equipped with up-to-date signatures. These results clearly confirm the need for alternative detection techniques, as provided by Cujo and Jsand, for successfully defending against the threat of drive-by-download attacks.

False-positive rates. Table 3 shows the false-positive rates on the benign data sets for all detection methods. Except for AntiVir all methods attain reasonably low false-positive rates. The combined analysis of Cujo, for example, yields a false-positive rate of 0.004%, corresponding to 4 false alarms in 100.000 visited web sites, on the Alexa-200k data set.

Benign data sets		Cujo		ClamAv	AntiVir	Jsand
	static	dynamic	combined			
Alexa-200k	0.003%	0.001%	0.004%	0.000%	0.087%	—
Surfing	0.000%	0.000%	0.000%	0.000%	0.000%	—
Cova et al. [4]	—	—	—	—	—	0.013%

Table 3: Comparison of false-positive rates on benign data sets. Results for Cujo have been averaged over 10 independent runs. Results for Jsand have been taken from Cova et al. [4].

The high false-positive rate of AntiVir with 0.087% is mainly due to over generic detection rules. The majority of false alarms shows the label `HTML/Redirector.X`, indicating a potential redirect, where the remaining alerts have generic labels, such as `HTML/Crypted.Gen` and `HTML/Downloader.Gen`. We verified each of these alerts using a client-based honeypot [19], but could not determine any malicious activity.

For the false alarms raised by Cujo, we identify two main causes: 0.003% of the web pages in the Alexa-200k data set contain fully encrypted JavaScript code with no plain-text operations except for `unescape` and `eval`. This drastic form of obfuscation induces the false alarms of our static analysis. The 0.001% false positives of the dynamic analysis result from web pages redirecting error messages of JavaScript to customized functions. Though applied in a benign context in these 0.001% cases, such redirection is frequently used in drive-by downloads to hide errors during exploitation of vulnerabilities.

Overall, this experiment demonstrates the excellent detection performance of Cujo which identifies the vast majority of drive-by downloads with very few false alarms—although all attacks have been unknown to the system. Cujo thereby significantly outperforms current anti-virus tools and is almost on par with the offline analysis system Jsand.

3.2 Explanations

After studying the detection performance of Cujo, we explore its ability to equip alerts with appropriate explanations, which provides a valuable instrument for further analysis of detected attacks in practice. Due space constraints, we herein focus on explanations of dynamic analysis, though similar results can also be generated for static analysis of Cujo.

Figure 4(a) shows the top five features (3-grams) with largest contribution to detection of a heap spraying attack, as described in Section 2.3. The attack type is clearly manifested in all five features. The first feature corresponds to the abstract operation `HEAP SPRAYING DETECTED` which is triggered by our sandbox and indicates unusual memory activity. The remaining features reflect typical patterns of a shellcode construction, including the unescaping of an encoded string along with a corresponding NOP sled.

$\phi_s(x) \cdot w_s$	**Features** (3-grams)
0.190	`HEAP SPRAYING DETECTED`
0.121	`CALL unescape SET`
0.053	`SET global.shellcode TO`
0.053	`unescape SET global.shellcode`
0.036	`TO "%90%90%90%90%90%90%90...`

(a) Dynamic features of a heap spraying attack

$\phi_s(x) \cdot w_s$	**Features** (3-grams)
0.036	`CALL unescape CALL`
0.030	`CALL fromCharCode CALL`
0.025	`CALL eval CONVERT`
0.024	`parseInt CALL fromCharCode`
0.024	`CALL createElement ("object")`

(b) Dynamic features of an obfuscated attack

Figure 4: Examples for explanation of dynamic detection. The five features with highest contribution to the dynamic detection are presented for two drive-by-download attacks.

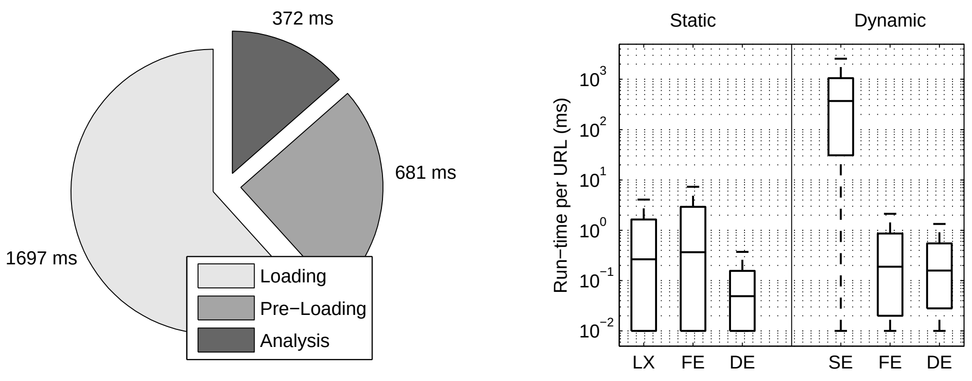
A second example for explanations of a detection is presented in Figure 4(b), which shows the top five features of a drive-by-download attack with strong obfuscation. Several calls to functions typical for obfuscation and corresponding substitution ciphers are visible, including `eval` and `unescape` as well as the conversion functions `parseInt` and `fromCharCode` used during decryption of the attack. The last feature reflects the instantiation of an object likely related to a vulnerability in a browser extension, though the actual details of this exploitation are not covered by the first five features.

It is important to note that the explanations of Cujo are specific to the detection of particular attacks and must not be interpreted as stand-alone detection rules. While we have only shown the top five features for explanations, the underlying detection model of Cujo involves 8 million features for dynamic analysis and thus realizes a complex decision function, hard to explain in a general manner.

3.3 Run-time Performance

Given the accurate detection of drive-by downloads, it remains to show that Cujo provides sufficient run-time performance for practical application. Hence, we first examine the individual run-time of each system component individually and then study the overall processing time in a real application setting with multiple users. All run-time experiments are conducted on a system with an Intel Core 2 Duo 3 GHz processor and 4 Gigabytes of memory.

Run-time of components. For the first analysis, we split the total run-time of Cujo to the contributions of individual components as depicted in Figure 1. For this, we add extra timing information to the JavaScript analysis, the feature extraction and learning-based detection. We then measure the exact contributions to the total run-time on a sample of 10,000 URLs from the Alexa-200k data set.



(a) Median run-time per URL (b) Statistical breakdown of run-time

Figure 5: Run-time performance of Cujo. Loading and analysis times have been measured on a sample of 10.000 URLs from Alexa-200k data set. Analysis time divides into lexing JavaScript (LX), sandbox emulation (SE), feature extraction (FE) and detection (DE).

Figure 5(a) shows the median run-time per URL in milliseconds, including loading of a web page, pre-loading of external JavaScript code and the actual analysis of Cujo. Surprisingly, most of the time is spent for loading and pre-loading of content, whereas only 14% is devoted to the analysis part of Cujo. As we will see in the following section, we can greatly benefit from this imbalance by employing regular caching techniques.

A detailed statistical breakdown of the analysis run-time is presented in Figure 5(b), where the distributions of run-time per URL are plotted for the static and dynamic analysis separately.

Each distribution is displayed as a boxplot, in which the box itself represents 50% of the data and the lower and upper markers the minimum and maximum run-time per URL. Additionally, the median is given as a middle line in each box. Except for the sandbox emulation, all components induce a very small run-time overhead ranging between 0.01 and 10 ms per URL. The sandbox analysis requires a median run-time of 370 ms per URL which is costly but still significantly faster then related sandbox approaches.

Operating run-time. In this experiment, we empirically evaluate the total run-time of Cujo in a real application setting. In particular, we deploy Cujo as a web proxy and measure the time required per delivery of a web page. To obtain reproducible measurements, we use the URLs in the Surfing data set as basis for this experiment, as it contains multiple surfing sessions of five individual users. For comparison, we also employ a regular web proxy, which just forwards data to the users. As most of the total run-time is spent for the loading and pre-loading of resources, we enable all caching capabilities in Cujo and the web proxy.

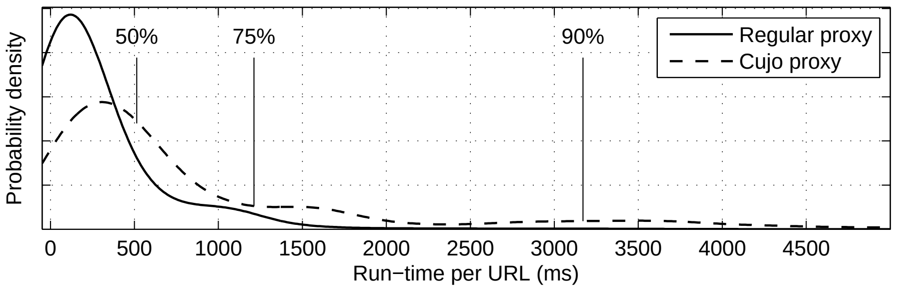


Figure 6: Operating run-time of Cujo and regular web proxy. The run-time has been measured on 20.000 URLs from Surfing data set. The URLs have been processed in their original order to take advantage of caching capabilities.

Results for this experiment are shown in Figure 6, where the distribution of run-time per URL is presented as a density plot. As expected the regular proxy proxy ranges in the front part of the plot with a median processing speed of roughly 150 ms per request. The run-time of Cujo is slightly shifted to the right in comparison with the regular proxy, yet its median lies around 500 ms per web page, thus inducing only a minimal delay at the web client. The run-time distribution of Cujo shows an elongated tail, where few web pages require more than 3,000 ms for processing due to excessive analysis of JavaScript code. We are currently experimenting with stopping the dynamic code analysis after 1,000 ms in these cases, though the limited analysis may negatively impact detection accuracy.

Overall, this experiment demonstrates that Cujo strongly benefits from caching capabilities, such that a median run-time of 500 ms can be attained. Although minor run-time peaks still exists, Cujo is actively used as secure web proxy in our institute for protecting (voluntary) web users from drive-by-download attacks.

4 Related Work

Since the first discovery of drive-by downloads, analysis and detection of this threat has been a vital topic in security research. One of the first studies on these attacks and respective defenses has been conducted by Provos et al. [15, 16]. The authors inspect web pages by monitoring a web browser for anomalous activity in a virtual machine. This setup allows for detecting a broad range of attacks. However, the analysis requires prohibitive run-time for on-line application, as the virtual machine needs to be restored and run for each web page individually.

A similar approach for identification of drive-by downloads is realized by client-based honeypots, such as Capture-HPC [19] and PhoneyC [14]. While Capture-HPC also relies on monitoring state changes in a virtual machine, PhoneyC emulates known vulnerabilities to capture attacks in a lightweight manner. Although effective in identifying web pages with malicious content, client-based honeypots are designed for offline analysis and thus suffer from considerable run-time overhead.

In contrast to these generic techniques, other approaches focus on identifying particular attacks types, namely heap spraying attacks. For example, the system Nozzle proposed by Ratanaworabhan et al. [17] intercepts the memory management of a browser for detecting valid x86 code in heap objects. Similarly, Egele et al. [8] instrument SpiderMonkey for scanning JavaScript strings for the presence of executable x86 code. Both systems provide an accurate and efficient detection of heap spraying attacks, yet they fail to identify other common types of drive-by downloads, for example, using insecure third-party extensions for infections.

Closest to our work is the analysis system Jsand developed by Cova et al. [4] as part of the Wepawet service. Jsand analyses JavaScript using the framework HtmlUnit and interpreter Rhino which enable emulating an entire browser environment and monitoring sophisticated interaction with the DOM tree. The recorded behavior is then analysed using 10 features specific to drive-by-download attacks for anomalous activity. Due to its public web interface, Jsand is frequently used by security researchers to study novel attacks and has proven to be a valuable analysis instrument. However, its broad analysis of JavaScript code is costly and induces a prohibitive average run-time of about 25 seconds per web page.

Finally, the system Noxes devised by Kirda et al. [13] implements a web proxy for preventing cross-site scripting attacks. Although not directly related to this work, Noxes is a good example of how a proxy system can transparently protect users from malicious web content. Obviously, this approach targets only cross-site scripting attacks and does not protect from other threats, such as drive-by downloads.

5 Conclusions

In this paper, we have presented Cujo, a system for effective and efficient prevention of drive-by downloads. As an extension to a web proxy, Cujo transparently inspects web pages using static and dynamic detection models and allows for blocking malicious code prior to delivery to the client. In an empirical evaluation with 200,000 web pages and 600 drive-by-download attacks, a prototype of this system significantly outperforms current anti-virus products and enables detecting 95% of the drive-by downloads with few false alarms and a median run-time of 500 ms per web page—a delay hardly perceived at the web client

While the proposed system does not generally eliminate the threat of drive-by downloads, it considerably raises the bar for adversaries to infect client systems. To further harden this defense, we currently investigate combining Cujo with offline analysis and honeypot systems. For example, malicious code detected using honeypots might be directly added to the training data of Cujo for keeping detection models up-to-date. Similarly, offline analysis might be applied for inspecting and explaining detected attacks in practice.

References

[1] Standard ECMA-262: ECMAScript Language Specification (JavaScript). 3rd Edition, ECMA International, 1999.

[2] Symantec Global Internet Security Threat Report: Trends for 2009. Vol. XIV, Symantec, Inc., 2010.

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

[4] 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.

[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)*, 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)*, 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*, 9:1871–1874, 2008.

[10] P. Fogla and W. Lee. q-gram matching using tree models. *IEEE Transactions on Knowledge and Data Engineering*, 18(4):433–447, 2006.

[11] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Proc. of IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, CA, USA, 1996.

[12] M. Johns. On JavaScript malware and related threats – Web page based attacks revisited. *Journal in Computer Virology*, 4(3):161–178, 2008.

[13] E. Kirda, C. Kruegel, G. Vigna, , and N. Jovanovic. Noxes: A client-side solution for mitigating cross site scripting attacks. In *Proc. of ACM Symposium on Applied Computing (SAC)*, 2006.

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

[15] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All your `iframes` point to us. In *Proc. of USENIX Security Symposium*, 2008.

[16] 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.

[17] 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.

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

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

[20] K. Wang, J. Parekh, and S. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Recent Adances in Intrusion Detection (RAID)*, pages 226–248, 2006.