Detection of Malicious Web Pages Using System Calls Sequences

Gerardo Canfora[1], Eric Medvet[2], and Francesco Mercaldo[1], and Corrado Aaron Visaggio[1]

[1] Dept. of Engineering, University of Sannio, Benevento, Italy
[2] Dept. of Engineering and Architecture, University of Trieste, Italy

Abstract. Web sites are often used for diffusing malware; an increasingly number of attacks are performed by delivering malicious code in web pages: drive-by download, malvertisement, rogueware, phishing are just the most common examples. In this scenario, JavaScript plays an important role, as it allows to insert code into the web page that will be executed on the client machine, letting the attacker to perform a plethora of actions which are necessary to successfully accomplish an attack. Existing techniques for detecting malicious JavaScript suffer from some limitations like: the capability of recognizing only known attacks, being tailored only to specific attacks, or being ineffective when appropriate evasion techniques are implemented by attackers. In this paper we propose to use system calls to detect malicious JavaScript. The main advantage is that capturing the system calls allows a description of the attack at a very high level of abstraction. On the one hand, this limits the evasion techniques which could succeed, and, on the other hand, produces a very high detection accuracy (96%), as experimentation demonstrated.

1 Introduction

In recent years, the web applications became an important vector of malware, as many reports state [1, 2]. A number of attacks are performed leveraging malicious web sites: drive-by-download, which consists of downloading and installing or running malware on the machine of the victim; csrf, which deviates the victim's navigation on a malicious web site; phishing, web sites which reproduce existing benign sites for obtaining credentials or other sensitive information from the victim; malvertisement, which is advertisement containing malware; malware serving, which collects traffic with different techniques and hosts malware; and rogueware, which is a fake antivirus which realizes illegal tasks, like stealing information or spying victim's machine.

Existing techniques may be very efficient in identifying specific and well-known attacks [3], but they often fail in detecting web threats which are new or scarcely diffused [4]. Since the attackers know that the approaches for detecting attacks are usually successful only against some kinds of attacks, different combinations of attacks are mixed together in order to evade detection [3].

Furthermore, the turbulent evolution of web technology entails a parallel evolution of web threats, which makes ineffective all those detecting techniques

S. Teufel et al. (Eds.): CD-ARES 2014, LNCS 8708, pp. 226–238, 2014.

which are strictly based on the technology of the web pages, because they observe the behavior of only certain components or characteristics, neglecting others which may be exploited by attackers. The introduction of HTML5, for instance, is bearing new functions like inline multimedia and local storage, which could be leveraged for designing new attacks [5], whose dynamics are not expected by current detectors.

A system for circumventing these hurdles is to analyze the web threats at a finer grain, which is the one of the operating system. The conjecture which we aim to demonstrate with this paper is that observing the behavior of a web application as the sequence of system calls invoked by the system when the browser connects to the web application makes the capability of detection independent of the specific web threat. This should result in a more effective detection system, which is able to exhibit a higher accuracy, i.e., reduce the number of false negatives and false positives. To observe the behavior of a web application at the level of system calls means decomposing at the smallest units of computation, or rather obtaining a very high level of abstraction of the code features.

The conjecture relies on the idea that a web application designed for performing an attack instead of performing some specific (and benign) business logic should show characteristics, in terms of sequence of system calls, which are common to many attacks, independently from the type of attack and its implementation. For instance, a malicious web application is often hosted on web server with poorer performances (often due to the fact that a malicious web application must frequently change hosting server because these servers are blacklisted) than a benign web application (which needs high speed server for business reasons). Moreover, benign web applications have a more complex structure than malicious web application, whose only purpose is to perform attacks and not to provide business services, several functions to user, or many pieces of information. This could cause, for instance, a fewer number of `open` system calls invoked by the malicious web application, than by the benign ones.

We wish to investigate whether malicious web applications and benign web applications differ in terms of the system calls they invoke. Thus, we pose two research questions:

- RQ1: is there a significant difference in the occurrences of system calls invoked by malicious and by trusted web applications?
- RQ2: are there sequences of system calls which are more frequent in malicious web applications than in trusted web applications?

RQ1 aims at verifying whether malicious web applications have system calls with different occurrences of benign web applications. A similar finding was observed for malware, where some op-codes had a larger or smaller number of occurrences than in non-malware code [6]. RQ2 consists of exploring the possibility that specific sequences of system calls could characterize malicious web applications, i.e., are more (or less) frequent in malicious web applications than in benign ones. As data analysis demonstrates, both RQ1 and RQ2 have a positive answer.

The paper proceeds as follows: Section 2 analyses the related literature, Section 3 and 4 discuss experimentation and results obtained, respectively for RQ1 and RQ2, and, finally, Section 5 draws the conclusions.

2 Related Work

A wealth of techniques exist to detect, prevent or characterize malicious activities carried on using web pages.

Blacklists collect malicious URLs, Ip addresses, and domain names obtained by manual reporting, honeyclients, and custom analysis service [3]. Blacklisting requires trust by the users and a huge management effort for continuously updating the list and verifying the dependability of the information.

Heuristic-based techniques [7, 8] leverage signatures of known malicious codes: if a known attack pattern is found within a web application, it is flagged as malicious. However, signatures can be successfully evaded by malicious code—obfuscation being the most commonly used technique. Moreover, this mechanism is not effective with unknown attacks.

Static Analysis techniques [9–12, 8, 13, 14] extract features from URL string, host identity, HTML, JavaScript code, and reputation metadata of the page. These values are entered in machine learning based classifiers which decide whether the web application is malicious or not. Obfuscated JavaScript, exploiting vulnerabilities in browser plug-ins and crafted URLs are common practices to evade this form of detection.

Dynamic analysis [15–18, 13] observes the execution of the web application. Proxy-level analysis [19] captures suspicious behaviors, such as unusual process spawning, and repeated redirects. Sandboxing techniques [7, 20] produce a log of actions and find for known patterns of attacks or unsual sequence of actions. Honeyclients [21] mimic a human visit in the website, but by using a dedicated sandbox. Execution traces and features are collected and analyzed to discover attacks. Low-interaction honeyclients [22] compare the execution traces with a set of signatures, which makes this approach ineffective with zero-day attacks. High-interaction honeyclients [23–25] look for integrity changes of the system states, which means monitoring registry entries, file systems, processes, network connections, and physical resources (memory, CPU). Honeyclients are powerful, but at a high computational cost, as they need to load and execute the web application. Honeyclients are useless for time-based attacks, and, moreover, malicious server can blacklist honeyclients IP address, or they can be discovered by using Turing Test with CAPTCHAs [3].

Different methods have been proposed for detecting and analyzing malicious Java script code. Zozzle [26] extracts features of context from AST, such as specific variable names or code structure. Cujo [20] obtains q-grams from the execution of JavaScript and classifies them with machine learning algorithms. Code similarity is largely used to understand whether a program is malicious or not, by comparing the candidate JavaScript with a set of known malicious JavaScripts. Revolver [27] computes the similarity by confronting the AST structure of two JavaScript pairs.

Clone detection techniques have been proposed in some papers [28, 29], but they assume that the programs under analysis do not show an adversarial behavior. Such an assumption does not hold when analyzing malicious programs. Attackers usually change the code corresponding to the payload taken form other existing malware for evading clone detection. Another strong limitation is the large number of source code the candidate code sample must be compared with. Bayer et al. [30] intend to solve this problem by leveraging locality sensitive hashing, while Jong et al. [31] make use of feature hashing for reducing the feature space.

At the best knowledge of the authors the method we propose is new in the realm of malicious JavaScript detectors, and its main advantages are: the success is independent from the type of attack, and it is designed to be robust against evasion techniques, as discussed later in the paper.

3 RQ1: System Call Occurrences

3.1 Data Collection

We performed an experimental analysis aimed at investigating the point addressed by RQ1—i.e., whether a significant difference exists in the occurrences of system calls invoked by malicious and by trusted web applications. To this end, we composed a set including malicious and trusted web pages and recorded the system call traces which are generated while visiting them.

We chose at random more than 3000 URLs from the Malware Domain List[1] archive: this archive contains about 80 000 URLs of malicious web pages implementing different attacks patterns. We chose the first 3000 URLs included in the Alexa Global Top Sites[2] ranking for making up the set of trusted websites.

We then systematically visited each of these URLs and recorded the system calls traces. In order to collect system calls traces, we used Strace[3], which is a tool for Unix platform diagnostic and debugging. Strace hooks a running process and intercepts the system calls done by the process and register them within a log. Strace can be configured in two different modes: "verbose", which collects all the system calls of the target process with all the metadata, and "summary", which collects aggregated data, such as the number of calls for each system call, the total time required for the system call, the number of errors, the percentage of user time. We used Strace in "verbose" mode. We automated the collection procedure by means of a Java program which we built for performing the following steps—being u the URL for which the system call trace has to be collected:

1. launch Strace, configured to hook the Firefox process;
2. launch Firefox with u as the only URL to visit;
3. wait 60 s;
4. kill Firefox and Strace;
5. truncate the system call trace to the calls performed during the first 20 s.

[1] `http://www.malwaredomainlist.com`
[2] `http://www.alexa.com/topsites`
[3] `http://sourceforge.net/projects/strace/`

We executed the collection procedure for all the URLs in a row, running the Java program on a machine hosted in our campus, provided with good and stable connectivity to the Internet.

We ensured that no errors were generated while visiting URLs. We excluded all those traces for which one of the following abnormal situation occurred: HTTP response codes 302 and 404, "unable to resolve host address" error and "connection time out" error. We stopped the collection after visiting exactly 3000 malicious and 3000 trusted pages.

3.2 Analysis

We denote with N the length of a system call trace and by N_c the number of occurrences of the system call c within a trace.

We observed 106 different system calls. The average number $\overline{N}$ of system calls we collected for each URL was 114 000 and 76 710, respectively for trusted and malicious pages. We think that the difference is justified mainly by the different connectivity and processing power of servers serving trusted and malicious pages. In particular, malicious pages are often served by improvised web server—possibly compromised machines which were not meant to act as servers—and hence with bad connectivity and low processing power. Despite the number of system calls in the 20 s trace might appear a good indicator of a page being malicious, it cannot be actually used alone as a discriminant, since it strongly depends on the setting of the client: client connectivity and processing power were tightly controlled in our data collection settings, but are likely to be more variable in a real scenario.

Concerning the number of occurrences of system calls, Table 1 shows the absolute occurrences $\overline{N_c}$ and relative occurrences $\overline{\frac{N_c}{N}}$ of the 10 most occurring (considering all traces) system calls, for trusted and malicious pages: for example, the `futex` call occurs on the average 12 895 times in each trusted trace, which corresponds to 11.36% of calls per trace. As expected, the absolute number of occurrences is in general greater for trusted pages. In relative terms, figures are similar for trusted and malicious pages, with some exceptions. The call `gettimeofday` is by far the most occurring across malicious pages: on the average, more than $\frac{1}{3}$ of system calls are `gettimeofday`; this system call could be invoked for time-based attacks and logic bombs. On the other hand, for trusted pages, the most occurring call is `clock_gettime`. This is likely to happen because, for a time-based attack, a temporal grain at level of a day is enough as temporal line after which the attack must be launched. The call `clock_gettime` can be used by many common functions in current trusted websites: for instance in forums and social networks, the local clock time is commonly used to tag the published posts. Moreover the local clock time is used for web page dynamic updates (for instance this is typical in news websites), or for JavaScript timers bound to UI activieties, and so on. The call `open` seems to occur more frequently while visiting malicious web pages than trusted ones: 4.20% vs. 2.88%. This could

Table 1. Most occurrent system calls in our dataset

	Trusted		Malicious	
System call c	$\overline{N_c}$	$\overline{\frac{N_c}{N}}$ (%)	$\overline{N_c}$	$\overline{\frac{N_c}{N}}$ (%)
1 `clock_gettime`	35 444	29.97	17 869	21.81
2 `gettimeofday`	30 675	27.53	25 622	34.54
3 `futex`	12 895	11.36	9757	12.55
4 `recv`	9325	8.25	5105	6.63
5 `poll`	6919	6.13	4073	5.24
6 `open`	3108	2.88	3052	4.20
7 `read`	2336	2.07	1917	2.51
8 `writev`	2183	1.97	1231	1.64
9 `write`	1752	1.53	1206	1.57
10 `stat64`	1348	1.22	1149	1.54

be due to the fact that many kinds of attacks performed through web pages, as previously observed, consist of gathering private information from the victim's machine, obtaining the machine control, changing machine settings (e.g., DNS poisoning, registry modifications, password cracking, cookies, browser settings, and chronology retrieving), causing a denial of service, and so on.

Figure 1 shows the comparison between relative occurrences of the 10 most occurring system calls for trusted and malicious pages, by means of a boxplot. It can be seen that no one of the considered calls can be taken as a discriminant between trusted and malicious pages, because values for all the considered pages overlap for the two categories. For example, despite being the mean value for the `open` call significantly lower for trusted pages (see Table 1), several trusted pages (at most 25%) exceed the mean value of the `open` relative occurrence within malicious pages. This happens because nowadays many web applications must implement very complex business goals (e-health, e-government, e-banking, e-commerce) which require a complex architecture, rich of files (images, animations, CSS sheets, JavaScripts), and complex functions, which need to have access to cookies, Internet files (chronology), local folders (for updating files), web services. In this case the number of files which need to be opened can be remarkably higher than the ones opened by many web-based attacks.

Finally, we analyzed the 10 system calls for which the absolute difference $\Delta\overline{\frac{N_c}{N}}$ between the average relative occurrences in trusted and malicious page was the greatest. The rationale was to find those system calls which were the best candidate to be a good discriminant between trusted and malicious pages, regardless of being the calls rare or frequent. Table 2 lists the 10 system calls, along with the value of the difference in relative occurrences. It can be seen that 9 on 10 of the calls chosen with this procedure were also included in the set of the 10 most occurring calls (i.e., those of Table 1 and Figure 1): the only exception was the `_llseek` call which took the place of the `write` call.

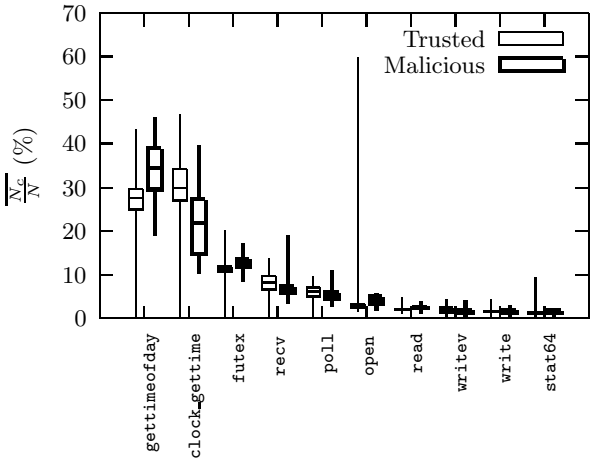

Fig. 1. Boxplot of relative occurrences of the 10 most occurring system calls in our dataset. The top and bottom edges of each box represent third and first quartile, respectively; the line inside the box represents the mean value; the vertical lines above and below each box span to max and min values, respectively.

Table 2. System calls with greatest difference in relative occurrences in our dataset

System call	$\Delta\overline{\frac{N_c}{N}}$ (%)
1 `clock_gettime`	8.16
2 `gettimeofday`	7.01
3 `recv`	1.62
4 `open`	1.32
5 `futex`	1.19
6 `poll`	0.89
7 `read`	0.44
8 `stat64`	0.35
9 `writev`	0.33
10 `_llseek`	0.31

3.3 Classification

We tried to exploit the difference in relative occurrences of system calls to build a classifier able to discriminate between trusted and malicious pages. To this end, we defined a method consisting of a training phase and a classification phase.

In the training phase, we proceed as follows. Let T be a set of labeled traces (t, l), where $l \in \{\text{trusted}, \text{malicious}\}$ is the label and t is the trace, i.e., a sequence of system calls. We build, for each trace t, a feature vector $\mathbf{f} \in [0, 1]^{106}$ composed of all the system call relative occurrences, sorted alphabetically on the calls themselves. Then, we train a Support Vector Machine (SVM) on the feature vectors using the labels in T. We use a third-degree polynomial kernel with cost parameter set to 1.

In the classification phase, we simply obtain the feature vector $\mathbf{f}$ from the trace under analysis and then applied the learned SVM.

We assessed the effectiveness of the proposed classification method on the dataset D collected as described in Section 3.1, with the following procedure:

1. built a training set $T \subset D$ by picking 2400 trusted traces and 2400 malicious traces;
2. built a testing set $T' = D \setminus T$;
3. run the training phase on T;
4. applied the learned classifier on each element of T'.

We performed a 5-fold cross validation, i.e., we repeated the four steps 5 times varying the composition of T (and hence of T').

We measured the performance in terms of accuracy, False Negative Rate (FNR) and False Positive Rate (FPR), i.e., respectively, the percentage of T' pages which were correctly classified, the percentage of malicious pages in T' which were wrongly classified as trusted and the percentage of trusted pages in T' which were wrongly classified as malicious.

We obtained a classification accuracy of 97.18%, averaged across the 5 repetitions, with a standard deviation $\sigma = 0.44\%$; FPR and FNR were respectively equal to 3.5% and 2.13%. Although such results are good, it is fair to note that a detector based on the number of invocations of a (set of) system calls could be evaded easily: if the number of malicious system calls is expected to be smaller than trusted ones, the attackers should write junk code which does not alter the payload effect but increases the number of those system calls. On the contrary, if the number of malicious system call is expected to be greater than trusted ones, as the value which we are considering is calculated in percentage, it is sufficient to increase the total number of all the system calls, with junk code, as well.

4 RQ2: System Calls Sequences

4.1 Classification

In order to answer RQ2, we considered a classification method for discriminating between trusted and malicious pages which bases on (short) system calls sequences, rather than occurrences. Similarly to the former case, we considered two phases: training and classification.

In the training phase, which operates on a set of labeled traces T, we proceed as follows. We first compute, for each trace, a feature vector $\mathbf{f}$ of n-gram occurrences (with $n = 3$). Each feature corresponds to ratio between the number of times a given subsequence of 3 system calls occurs in t and the number of 3-grams in t (which is $|t| - 2$). For example, if $t = \{\texttt{execve}, \texttt{brk}, \texttt{access}, \texttt{mmap2}, \texttt{access}, \texttt{open}, \texttt{stat64}, \texttt{open}, \texttt{stat64}, \texttt{open}\}$, then $f_{(\texttt{execve},\texttt{brk},\texttt{access})} = \frac{1}{8}$, $f_{(\texttt{brk},\texttt{access},\texttt{mmap2})} = \frac{1}{8}$, $\ldots$, $f_{(\texttt{open},\texttt{stat64},\texttt{open})} = \frac{2}{8}$, and so on.

The number $|C|^3$ of possible features is large, being C the set of system calls—recall that in our experimentation $|C| = 106$. For the sake of tractability, we consider only those features for which the corresponding 3-gram occurs at least once in T: this way, we reduce the number of features in our experimentation to $\approx 43\,000$.

Then, we perform a two-steps feature selection procedure. We first select the 5% of features with the greatest absolute difference between the average value computed only on trusted sequences and the average value computed only on malicious sequences. Second, among the remaining features, we select the k features with the highest mutual information with the label l.

Finally, we train a Support Vector Machine (SVM) on the selected features using the labels in T. We use a third-degree polynomial kernel with cost parameter set to 1.

In the classification phase, we simply extract the selected features from the trace under analysis and then apply the learned SVM. Note that the actual extraction of the features in this phase—including the collection of the trace itself—can be computationally cheaper, since only those k subsequences of system calls have to be counted.

4.2 Experimental Evaluation

We assessed the effectiveness of the proposed method on the same dataset and with the same procedure described in Section 3.3.

Table 3 shows the results of the experimental evaluation in terms of average value of accuracy, FNR and FPR across the 5 repetitions. It can be seen that our method is able to discriminate between trusted and malicious pages with an accuracy of 95.83% ($k = 25$): FNR and FPR are balanced, i.e., the method does not tend to misclassify one class of pages more than the other.

Moreover, results show that the best accuracy can be obtained with $k = 25$, but the method itself appears to be quite robust with respect to the parameter k. Considering that an actual implementation of our approach will benefit from low k values—since less data had to be recorded—the fact that the best accuracy can be obtained with $k = 25$ is a plus.

Results of Table 3 suggest that(i) the chosen features (3-gram occurrencies) are indeed informative for malicious pages detection, (ii) a large number of them do not provide any additional information—or the SVM classifier is not able to exploit it—and (iii) the proposed feature selection procedure allows to select the small fraction of features which allow an accurate classification.

For completeness of analysis, in Table 4 we show the 10 3-grams chosen with the feature selection procedure described in Section 4.1 in one repetition of our experimental evaluation—we verified that the list composition was stable across repetitions. It can be seen that the table includes system calls which were not captured by the criterion on call relative occurrences (see Table 1), nor by the criterion on difference in relative occurrences (see Table 2). This is the case of, e.g., `fstat64` and `set_robust_list`. This happens because some system calls fall in several sequences, like `write` , and `close`, so they are more

Table 3. Results in terms of accuracy, FNR and FPR

k	Accuracy (%)	FNR (%)	FPR (%)
10	94.52	5.27	5.70
25	95.83	4.23	4.10
50	95.33	4.70	4.63
100	94.55	5.50	5.40
250	94.63	5.47	5.27

Table 4. 3-grams of system calls chosen by the feature selection procedure in one repetition of our experimental evaluation

	3-gram of system calls
1	`clock_gettime`, `getdents`, `recv`
2	`write`, `send`, `getdents`
3	`gettimeofday`, `ioctl`, `mkdir`
4	`write`, `sendto`, `futex`
5	`close`, `write`, `connect`
6	`shutdown`, `recv`, `close`
7	`close`, `fstat64`, `set_robust_list`
8	`clock_gettime`, `setsockopt`, `recvmsg`
9	`open`, `getrusage`, `clock_gettime`
10	`mkdir`, `getsockname`, `setsockopt`

frequent than others which occur only in one or two sequences, like `getdents` and `shutdown`. This suggests that considering the occurrences of sequences of system calls allows to take into account behaviors—defined by short sequences—which characterize benign or malicious activities. In other words, the concept of system calls sequence encloses the concept of a program behavior at a very low-level grain and, at the same time, at a high level of abstraction with respect to the type of attacks and its implementation in the web application. Somehow, the sequences of system calls can be seen as fingerprints or signatures of malicious payload (at a very high level of abstraction). Conversely, occurrences of system calls are not as much clearly representative of an attack as the system calls sequences. In fact, occurrence counting is a too rough feature, as in the counting can be included system calls that are not used in the payload.

Summing up,(i) features considered by RQ2 produce results not significantly worse than those considered by RQ1 (the accuracy is about 96% and 97%, respectively), but (ii) the former appear to be less prone to be circumvented by trivial evasion techniques. As previously explained, the occurrence of a system call can be altered by adding some junk code, which is a relatively straightforward technique. On the contrary, to camouflage a system calls sequence is much harder, because the system calls sequence is a direct image at operating system of the malicious behavior. To define an altered system calls sequence, without affecting the intended payload, could be feasible, but very hard to realize. In fact, being the actual maliciuos code in JavaScript, insertion of junk JavaScript code will likely not impact on short system call sequences. In other words,

since the system calls sequence to be invoked depends on the effect which the code should produce, the insertion of junk code may add further system calls, but it cannot remove the sequence of system calls which represents the malicious behavior.

5 Conclusions and Future Work

With this paper we evaluate two methods for detecting malicious web pages based on the system calls which are invoked when the browser connects to the web application under analysis.

The first method consists of counting the occurrences of specific system calls, while the second method consists of retrieving specific sequences of system calls which are more frequent in malicious web applications than in trusted ones. Both the method produced a high classification accuracy, the first method exhibiting an accuracy slightly higher than the second one (97% vs. 96%). However, a detection method which only exploits the occurrences of specific system calls can be evaded easily, by adding junk code which alters the counting. On the contrary, altering a sequence of system calls is much harder, as it depends directly on the specific effect the attacker intends to realize. Adding junk code, in this case, may alter the number of system calls, but not a specific sequence, which may represent a malicious behavior.

As future work, we are planning to enlarge the experimentation by testing the proposed technique on a data with noise, i.e., with data collected by real clients during navigation—an in vivo experimentation. Additionally, we wish to investigate about methods for inferring a pattern for those specific sequences of system calls which correspond to malicious activities. In fact, the proposed method is capable to identify system calls sequences common to malicious web applications, but it is not able to map which ones correspond to which malicious effect.

References

1. 2013 threats predictions (2013), http://www.mcafee.com/us/resources/reports/rp-threat-predictions-2013.pdf
2. Pandalabs quarterly report: January - March 2013 (2013), https://www.switch.ch/export/sites/default/about/news/2013/files/PandaLabs-Quaterly-Report.pdf
3. Eshete, B.: Effective analysis, characterization, and detection of malicious web pages. In: Proceedings of the 22nd International Conference on World Wide Web Companion, pp. 355–360. International World Wide Web Conferences Steering Committee (2013)
4. Trend Micro: Web threats (2012), http://apac.trendmicro.com/apac/threats/enterprise/web-threats
5. Weiss, A.: Top 5 security threats in html5 (2011), http://www.esecurityplanet.com/trends/article.php/3916381/Top-5-Security-Threats-in-HTML5.htm

6. Canfora, G., Iannaccone, A.N., Visaggio, C.A.: Static analysis for the detection of metamorphic computer viruses using repeated-instructions counting heuristics. Journal of Computer Virology and Hacking Techniques, 11–27 (2013)
7. Dewald, A., Holz, T., Freiling, F.C.: Adsandbox: Sandboxing javascript to fight malicious websites. In: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 1859–1864. ACM (2010)
8. Seifert, C., Welch, I., Komisarczuk, P.: Identification of malicious web pages with static heuristics. In: Australasian Telecommunication Networks and Applications Conference, ATNAC 2008, pp. 91–96. IEEE (2008)
9. Canali, D., Cova, M., Vigna, G., Kruegel, C.: Prophiler: a fast filter for the large-scale detection of malicious web pages. In: Proceedings of the 20th International Conference on World Wide Web, pp. 197–206. ACM (2011)
10. Choi, H., Zhu, B.B., Lee, H.: Detecting malicious web links and identifying their attack types. In: Proceedings of the 2nd USENIX Conference on Web Application Development, p. 11. USENIX Association (2011)
11. Ma, J., Saul, L.K., Savage, S., Voelker, G.M.: Beyond blacklists: learning to detect malicious web sites from suspicious urls. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1245–1254. ACM (2009)
12. Ma, J., Saul, L.K., Savage, S., Voelker, G.M.: Identifying suspicious urls: an application of large-scale online learning. In: Proceedings of the 26th Annual International Conference on Machine Learning, pp. 681–688. ACM (2009)
13. Thomas, K., Grier, C., Ma, J., Paxson, V., Song, D.: Design and evaluation of a real-time url spam filtering service. In: 2011 IEEE Symposium on Security and Privacy (SP), pp. 447–462. IEEE (2011)
14. Sorio, E., Bartoli, A., Medvet, E.: Detection of hidden fraudulent urls within trusted sites using lexical features. In: 2013 Eighth International Conference on Availability, Reliability and Security (ARES), pp. 242–247. IEEE (2013)
15. Kim, B.-I., Im, C.-T., Jung, H.-C.: Suspicious malicious web site detection with strength analysis of a javascript obfuscation. International Journal of Advanced Science & Technology 26 (2011)
16. Ikinci, A., Holz, T., Freiling, F.: Monkey-spider: Detecting malicious websites with low-interaction honeyclients, sicherheit (2008)
17. Kolbitsch, C., Livshits, B., Zorn, B., Seifert, C.: Rozzle: De-cloaking internet malware. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 443–457. IEEE (2012)
18. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious javascript code. In: Proceedings of the 19th International Conference on World Wide Web, pp. 281–290. ACM (2010)
19. Moshchuk, A., Bragin, T., Deville, D., Gribble, S.D., Levy, H.M.: Spyproxy: Execution-based detection of malicious web content. In: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, vol. 3, pp. 1–16. USENIX Association (2007)
20. Rieck, K., Krueger, T., Dewald, A.: Cujo: efficient detection and prevention of drive-by-download attacks. In: Proceedings of the 26th Annual Computer Security Applications Conference, pp. 31–39 (2010)
21. Qassrawi, M.T., Zhang, H.: Detecting malicious web servers with honeyclients. Journal of Networks 6(1) (2011)
22. The honeynet project (2011), `https://projects.honeynet.org/honeyc`
23. Mitre honeyclient project (2011), `http://search.cpan.org/~mitrehc`

24. Capture-hpc client honeypot / honeyclient (2011), https://projects.honeynet.org/capture-hpc
25. Wang, Y.-M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., King, S.: Automated web patrol with strider honeymonkeys. In: Proceedings of the 2006 Network and Distributed System Security Symposium, pp. 35–49 (2006)
26. Curtsinger, C., Livshits, B., Zorn, B., Seifert, C.: Zozzle: Low-overhead mostly static javascript malware detection. In: Proceedings of the Usenix Security Symposium (2011)
27. Kapravelos, A., Shoshitaishvili, Y., Cova, M., Kruegel, C., Vigna, G.: Revolver: An automated approach to the detection of evasive web-based malware. In: USENIX Security Symposium (2013)
28. Pate, J.R., Tairas, R., Kraft, N.A.: Clone evolution: a systematic review. Journal of Software: Evolution and Process 25(3), 261–283 (2013)
29. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. School of Computing TR 2007-541, Queen's University (2007)
30. Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, behavior-based malware clustering. In: NDSS, vol. 9, pp. 8–11. Citeseer (2009)
31. Jang, J., Brumley, D., Venkataraman, S.: Bitshred: feature hashing malware for scalable triage and semantic analysis. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 309–320. ACM (2011)