Prophiler: A Fast Filter for the Large-Scale Detection of Malicious Web Pages

Davide Canali
Institute Eurecom, France
dcanali@iseclab.org

Marco Cova
University of Birmingham, UK
m.cova@cs.bham.ac.uk

Giovanni Vigna Christopher Kruegel
University of California, Santa Barbara
{vigna,chris}@cs.ucsb.edu

ABSTRACT

Malicious web pages that host drive-by-download exploits have become a popular means for compromising hosts on the Internet and, subsequently, for creating large-scale botnets. In a drive-by-download exploit, an attacker embeds a malicious script (typically written in JavaScript) into a web page. When a victim visits this page, the script is executed and attempts to compromise the browser or one of its plugins. To detect drive-by-download exploits, researchers have developed a number of systems that analyze web pages for the presence of malicious code. Most of these systems use dynamic analysis. That is, they run the scripts associated with a web page either directly in a real browser (running in a virtualized environment) or in an emulated browser, and they monitor the scripts' executions for malicious activity. While the tools are quite precise, the analysis process is costly, often requiring in the order of tens of seconds for a single page. Therefore, performing this analysis on a large set of web pages containing hundreds of millions of samples can be prohibitive.

One approach to reduce the resources required for performing large-scale analysis of malicious web pages is to develop a fast and reliable filter that can quickly discard pages that are benign, forwarding to the costly analysis tools only the pages that are likely to contain malicious code. In this paper, we describe the design and implementation of such a filter. Our filter, called *Prophiler*, uses static analysis techniques to quickly examine a web page for malicious content. This analysis takes into account features derived from the HTML contents of a page, from the associated JavaScript code, and from the corresponding URL. We automatically derive detection models that use these features using machine-learning techniques applied to labeled datasets.

To demonstrate the effectiveness and efficiency of *Prophiler*, we crawled and collected millions of pages, which we analyzed for malicious behavior. Our results show that our filter is able to reduce the load on a more costly dynamic analysis tools by more than 85%, with a negligible amount of missed malicious pages.

Categories and Subject Descriptors

K.4.1 [**Public Policy Issues**]: Abuse and crime involving computers; K.6.5 [**Security and Protection**]: Invasive software (e.g., viruses, worms, Trojan horses)

General Terms

Security, Design, Experimentation, Performance

WWW 2011, March 28–April 1, 2011, Hyderabad, India.
ACM 978-1-4503-0632-4/11/03.

Keywords

Malicious web page analysis, drive-by download exploits, efficient web page filtering

1. INTRODUCTION

The world wide web has become an integral part in the lives of hundreds of millions of people who routinely use online services to store and manage sensitive information. Unfortunately, the popularity of the web has also attracted miscreants who attempt to abuse the Internet and its users to make illegal profits.

A common scheme to make money involves the installation of malicious software on a large number of hosts. The installed malware programs typically connect to a command and control (C&C) infrastructure. In this fashion, the infected hosts form a botnet, which is a network of machines under the direct control of cyber criminals. As a recent study has shown [29], a botnet can contain hundreds of thousands of compromised hosts, and it can generate significant income for the botmaster who controls it.

Malicious web content has become one of the most effective mechanisms for cyber criminals to distribute malicious code. In particular, attackers frequently use drive-by-download exploits to compromise a large number of users. To perform a drive-by-download attack, the attacker first crafts malicious client-side scripting code (typically written in JavaScript) that targets a vulnerability in a web browser or in one of the browser's plugins. This code is injected into compromised web sites or is simply hosted on a server under the control of the criminals. When a victim visits a malicious web page, the malicious code is executed, and, if the victim's browser is vulnerable, the browser is compromised. As a result, the victim's computer is typically infected with malware.

Drive-by-download attacks have become pervasive over the last few years, and real-world examples show that legitimate (and presumably well-maintained) web sites are frequently compromised and injected with malicious code [7, 8].

Given the rising threat posed by malicious web pages, it is not surprising that researchers have started to investigate techniques to protect web users. Currently, the most widespread protection is based on URL blacklists. These blacklists (such as Google Safe Browsing) store URLs that were found to be malicious. The lists are queried by a browser before visiting a web page. When the URL is found on the blacklist, the connection is terminated or a warning is displayed. Of course, to be able to build and maintain such a blacklist, automated detection mechanisms are required that can find on the Internet web pages containing malicious content.

The tools of choice for the identification of malicious web pages are (high-interaction) honeyclients. These honeyclients, such as the MITRE HoneyClient [13], Microsoft's HoneyMonkey [30], Capture-HPC [25], or Google Safe Browsing [22], run a web browser on a real operating system inside a virtual machine. The browser is pointed to a URL that should be analyzed. After the corresponding page is loaded, the honeyclient system checks for artifacts that indicate a successful attack, such as executable files on the file system or unexpected processes. While the presence of such artifacts is strong evidence that a page is malicious, the drawback of high-interaction honeyclients is the fact that the analysis is expensive. While parallelization can help in processing multiple pages more efficiently, still the HTML page needs to be rendered and active content (such as JavaScript) needs to be executed. Moreover, after each successful exploit, the virtual machine needs to be restored, since the analysis platform can no longer be trusted. As a result, the analysis of a single URL can easily require several minutes.

In addition to high-interaction honeyclients, researchers have proposed alternative detection approaches for malicious web pages. In particular, a number of tools were proposed (such as Wepawet [4], PhoneyC [20], JSUnpack [15]) that rely on instrumented JavaScript run-time environments to detect the execution of malicious scripts, or only a certain kind of attacks (such as NOZZLE [23], a tool for the detection of heap-spraying on malicious web pages). Compared to high-interaction honeyclients, these systems provide more insights into the inner working of malicious scripts, and they require less effort to configure with a wide range of vulnerable plugins. However, they are not substantially faster, with analysis times ranging from seconds to a few minutes for a single page [4].

Unfortunately, the analysis time directly limits the scalability of these systems. As a result, it becomes very costly (if not impossible) to analyze millions of URLs in a day. This is problematic, both for organizations that try to maintain blacklists with good coverage (such as Google), but also, more generally, for everyone whose goal is to obtain a detailed and broad understanding of the malicious activity on the Internet with limited analysis resources.

One approach to address the limited scalability of current analysis systems is to devise an efficient filter that can quickly discard benign pages. By using such a filter as a front-end to a more sophisticated but resource-intensive back-end analysis system, one could save a large amount of resources, since the costly (but precise) back-end analysis is performed only on those pages that are likely to contain malicious content. Of course, one should be able to tune the sensitivity of the filter depending on the available analysis capacity and the acceptable level of false negatives (missed detections). In this context, false positives are less critical because even though they result in a waste of resources (that is, benign pages are analyzed using costly procedures), they are not increasing the exposure of users to threats.

In this paper, we present the design and implementation of a filtering system, called *Prophiler*, to quickly distinguish between likely malicious and likely benign web pages. *Prophiler* statically analyzes features of the HTML page, of the embedded JavaScript code, and of the associated URL using a number of models that are derived using supervised machine-learning techniques. Pages that are found to be likely malicious by *Prophiler* can then be further analyzed with one of the more in-depth (and costly) detection tools, such as Wepawet.

Since the web page being analyzed is not rendered and no scripts are executed, the analysis is fast. Compared to previous work that attempts to detect malicious web pages based on page content, our analysis uses a significantly more comprehensive set of features, and, as a result, delivers more precise results. Researchers have also suggested identifying malicious pages based on features extracted from URLs alone. This approach delivers good results for scam and phishing pages, since the corresponding URLs are often crafted by attackers to mislead users. However, when malicious content (such as a drive-by-download exploit) is injected into a legitimate page, the URL is not affected. Hence, systems based exclusively on URL features suffer from a substantial amount of false negatives, as shown in our experiments.

The need for a fast filter to enable the large-scale analysis of malicious web pages was previously recognized by Provos et al. [22] (some of the authors are also involved in Google's Safe Browsing efforts). Unfortunately, for obvious reasons, very few details have been revealed about Google's filter. In particular, the authors only provide examples of three page features and report that they use a proprietary machine-learning framework. Of course, the existence of Google's blacklist provides evidence that the overall system (combining the filter with the back-end analysis tools) works. Nevertheless, we feel that there are significant benefits in describing the technical details of our filtering approach in the literature: First, we introduce a comprehensive set of page and URL features for identifying malicious web pages. This allows others to build similar filters, making better use of their available analysis resources. Second, we discuss the trade-offs between false negatives and false positives, and we compare the performance of our filter to a number of previous systems. Third, we demonstrate that our filter allows us to dramatically improve the scale of the analysis that can be performed in the case of a publicly-available system, called Wepawet.

2. RELATED WORK

In the last few years, the detection of web pages that launch drive-by-download attacks has become an active area of research and several new approaches have been proposed.

Dynamic approaches. Dynamic approaches use honeyclient systems to visit web pages and determine if they are malicious or not. In high-interaction honeyclients, the analysis is performed by using traditional browsers running in a monitored environment and detecting signs of a successful drive-by-download attack (e.g., changes in the file system, the registry, or the set of running processes) [18,22,25,30]. In low-interaction honeyclients, the analysis relies on emulated browsers whose execution during the visit of a web page is monitored to detect the manifestation of an attack (e.g., the invocation of a vulnerable method in a plugin) [4, 20].

Both high- and low-interaction systems require to fully execute the contents of a web page. This includes fetching the page itself, all the resources that are linked from it, and, most importantly, interpreting the associated dynamic content, such as JavaScript code. These approaches usually yield good detection rates with low false positives, since, by performing dynamic analysis, they have complete "visibility" into the actions performed by an attack. The down-side is that this analysis can be relatively slow, because of the time required by the browser (either simulated or real) to retrieve and execute all the contents comprising a web page, taking from a few seconds to several minutes, depending on the complexity of the analyzed page.

Scalability issues with today's honeyclient systems (relatively slow processing speed combined with relatively high hardware requirements) motivated our work on a filtering system. Our filter achieves higher performance by forgoing dynamic analysis (e.g., the interpretation of JavaScript code), and relying instead on static analysis only.

Static approaches. Static approaches to the detection of drive-by-download attacks rely on the analysis of the static aspects of a web page, such as its textual content, features of its HTML and JavaScript code, and characteristics of the associated URL. Table 1 compares our approach to other relevant work in this area, in terms of the features used to evaluate web pages.

Class of features	Number of features					
	Prophiler	[27]	[16]	[5]	[17]	[26]
HTML	19	5	0	0	0	0
JavaScript	25	3	16	4	0	0
URL	12	0	0	0	4	0
Host	21	0	0	0	16	6
Total	77	8	16	4	20	6

Table 1: Comparison of the features, divided in four different feature classes, considered by our work and by the related approaches.

String signatures (i.e., string patterns that are common in malicious code) are used by traditional antivirus tools, such as ClamAV [2], to identify malicious pages. Unfortunately, signatures can be easily evaded using obfuscation. Therefore, these tools suffer from high false negatives rates (earlier studies report between 65% and 80% missed detections [4, 24]), which make them unsuitable for filtering likely malicious pages. Our filter is also based on static techniques, but it achieves better detection rates by relying on a combination of several characteristics of a web page based on its HTML content, JavaScript code, and other URL and host features, rather than simple static string patterns. Moreover, our filter can be more aggressive in labeling a page as malicious. The reason is that incorrect detections are discarded by the subsequent (dynamic) back-end analysis, and hence, false positives only incur a performance penalty.

Several systems have focused on statically analyzing JavaScript code to identify malicious web pages [5,16,27]. The most common features extracted from scripts are the presence of redirects (e.g., assignments to the *location.href* property), the presence of functions commonly used for obfuscation/deobfuscation (such as *fromCharCode()*), calls to the *eval()* function, large numbers of string manipulation instructions, abnormally long lines, and the presence of shellcode-like strings. In our filter, we considerably extend the set of JavaScript features used for detection: for example, we detect the presence of sections of code resembling deobfuscation routines, we take into consideration the entropy of both the entire script and of the strings declared in it, we identify the number of event attachments, and we analyze both Document Object Model (DOM) manipulation functions and fingerprinting functions (such as *navigator.userAgent()*).

Seifert et al. [27] also use the characteristics of the HTML structure of a web page as indicators of maliciousness. For example, they consider the visibility and size of `iframe` tags and the number of script tags referencing external resources. We extend this analysis by adding more than ten new features, such as the number of out-of-place elements (e.g., scripts outside `<html>` tags), as well as the percentage of the page occupied by JavaScript code.

Characteristics of URLs and host information have been used in the past to identify sites involved in malicious activity, such as phishing and scams. Garera et al. use statistical techniques to classify phishing URLs [6]. Ma et al. [17] use lexical properties of URLs and registration, hosting, and geographical information of the corresponding hosts to classify malicious web pages at a larger scale. Later in the paper, we discuss the issues involved in applying this approach to detecting pages involved in drive-by-downloads (opposed to threats such as phishing and scam pages), and propose a number of new features that are effective in this context.

It is important to observe that we did not simply introduce new detection features for the sake of being able to point to a longer feature list. As our experiments demonstrate, adding these additional features significantly contributes to the improved accuracy of our system.

Several of the detection tools described here have been used as components of crawler-based infrastructures designed to effectively find malicious web pages, e.g., [14, 18, 22]. In Section 4, we describe a similar setup, where *Prophiler* is used as a fast filtering component.

3. APPROACH

The goal of *Prophiler* is to classify pages collected by a web crawler as either likely malicious or likely benign, i.e., as likely to launch a drive-by-download attack or not, respectively. To perform this classification task, *Prophiler* uses a set of models that evaluate the features extracted from a page. These models are derived using supervised machine-learning techniques. In the following, we first describe the features extracted from a web page, and then we discuss how models are derived.

3.1 Features

The features extracted from a web page are the basis to determine if a page is malicious or not. Because, by design, our filter does not execute any code associated with the web page, the collected features are derived statically. By doing this, it is possible to perform the analysis faster than in the case of currently-used dynamic approaches.

We inspect two main sources of information for features: the page's contents (both its HTML and JavaScript code) and the page's URL (both its lexical and host-based characteristics). Some of the features we use have been proposed before (either for the detection of drive-by-download attacks or for the identification of other threats, such as phishing). In this work, we introduce and evaluate 48 new features specifically designed for identifying pages involved in drive-by-download attacks.

3.1.1 HTML features

HTML features are based both on statistical information about the raw content of a page (e.g., the page length or the percentage of whitespaces) and on structural information derived from parsing the HTML code (e.g., the location of specific elements in the page). To parse HTML pages, we use the Neko HTML Parser [3] because of its versatility and speed in parsing HTML code. Since some of our features detect anomalies in the structure of a web page (e.g., out-of-place tags), which are silently corrected by Neko, we also parse web pages with HTMLparser [21], which performs no error correction. We do not currently extract features from CSS files, even though some exploits rely on malicious style sheets. This is left as future work.

More precisely, we extract the following 19 features from HTML content: the number of `iframe` tags, the number of hidden elements, the number of elements with a small area, the number of script elements (both included via the `src` attribute, and inline), the presence of scripts with a wrong file name extension, the percentage of scripting content in a page, the percentage of unknown tags, the number of elements containing suspicious content, the number of suspicious objects, the percentage of whitespace in the page, the presence of `meta` refresh tags, the number of `embed` and `object` tags, the number of elements whose source is on an external domain, the number of out-of-place elements, the number of included URLs, the presence of double documents, the number of known malicious patterns, and the number of characters in the page.

These features capture characteristics that are commonly found in pages that have been compromised: the presence of injected content pointing to external domains, the use of obfuscation, and the presence of side effects (e.g., out-of-place tags) of the attacks used to compromise a web page. Notice that some of these features are particularly difficult to evade for an attacker. For example, in SQL injection attacks (which are often used to inject malicious content in vulnerable web pages), attackers do not generally have complete control of the resulting page, and, as a consequence, cannot avoid the anomalies (such as malformed documents or repeated tags) that are detected by our features. Most of the features are self-explanatory. Below, we discuss some of the features that require additional discussion.

Number of elements with small area. Most of the elements used to carry out a drive-by-download infection are hidden, on purpose, by the attacker. However, most drive-by-download exploits do not use visibility attributes to hide their elements, and instead set explicitly the width and height of the elements used to deliver the attack to very small values. So, we included a feature that records the number of elements of type `div`, `iframe`, or `object`, whose dimension is less then a certain threshold (30 square pixels for the area, or 2 pixels for each side).

Number of elements containing suspicious content. This feature takes into account the number of elements whose content is "suspicious," i.e., the content between the start tag and the end tag of the element could be shellcode. We consider this content to be suspicious if it is longer than a certain threshold (128 characters) and contains less than 5% of whitespace characters. Note that we could use more sophisticated techniques to determine if specific content represents executable shellcode, but, in this case, we prioritize performance over precision.

Number of suspicious objects. Suspicious objects are `object` elements that are included in the document and whose `classid` is contained in a list of ActiveX controls known to be exploitable. This list is taken from the PhoneyC tool [20] and has been expanded with a number of other ActiveX controls commonly found in real-world exploits.

Number of included URLs. This feature counts the number of elements which, being not inline, are included specifying their source location. Elements such as `script`, `iframe`, `frame`, `embed`, `form`, `object` are considered in computing this feature, because they can be used to include external content in a web page. The `img` elements and other elements are not considered, as they cannot be used to include any executable code.

Number of out of place elements. This feature counts the number of elements that reside out of their natural positioning in the HTML document. This feature is useful to detect web pages that have become malicious as the result of a stored XSS or SQL injection attack. In these cases, it is common to see scripts or `iframes` included in strange positions, such as between `title` tags or after the end of the document (outside the `body` or `html` elements). `iframe, frame, form, script, object` and `embed` element positions are checked according to the allowed positioning, as defined by the HTML DTD specifications.

Presence of double documents. This feature indicates whether a web page contains two or more `html`, `head`, `title`, or `body` elements. This is not allowed by the HTML specification, but can be seen in certain malicious web pages as a side-effect of the compromise of a web site.

Number of known malicious patterns. This feature counts the number of occurrences of specific patterns commonly found in drive-by-download campaigns. The pattern list is compiled and updated by a human analyst. We currently identify only one of such patterns: the presence of a `meta` tag that causes the refresh of the page, pointing it to `index.php?spl=`, as this is very common in pages redirecting to exploit servers.

Prophiler extracts also a hash of the content of every HTML document (namely, an MD5 hash of the page), to avoid analyzing again a page that has already been analyzed, as well as a signature of the structure of the document, i.e., a signature of the tree representing its Document Object Model. This signature is used to determine if the page has a structure similar to one or more pages that have been analyzed before and determined to be malicious. If a match is found, the page is considered potentially malicious, and sent to the dynamic analysis tool.

3.1.2 JavaScript features

JavaScript features result from the static analysis of either a JavaScript file (such as the ones commonly served with a content type of `text/javascript` and similar), or of each script included in a web page via an inline `<script>` element. As for the HTML features, JavaScript features are both statistical and lexical.

Most malicious JavaScript scripts are obfuscated and packed, to make their analysis difficult. In some cases, malware authors adopt encryption schemes and techniques to prevent code debugging. To detect these characteristics, we implemented the extraction of some statistical measures (such as string entropy, whitespace percentage, and average line length). We also consider the structure of the JavaScript code itself, and a number of features are based on the analysis of the Abstract Syntax Tree (AST) extracted using the parser provided by Rhino [19]. For example, we analyze the AST of the code to compute the ratio between keywords and words, to identify common decryption schemes, and to calculate the occurrences of certain classes of function calls (such as *fromCharCode()*, *eval()*, and some string functions) that are commonly used for the decryption and execution of drive-by-download exploits.

We extract a total of 25 features from each piece of JavaScript code: the number of occurrences of the *eval()* function, the number of occurrences of the *setTimeout()* and *setInterval()* functions, the ratio between keywords and words, the number of built-in functions commonly used for deobfuscation, the number of pieces of code resembling a deobfuscation routine, the entropy of the strings declared in the script, the entropy of the script as a whole, the number of long strings, the maximum entropy of all the script's strings, the probability of the script to contain shellcode, the maximum length of the script's strings, the number of long variable or function names used in the code, the number of string direct assignments, the number of string modification functions, the number of event attachments, the number of fingerprinting functions, the number of suspicious objects used in the script, the number of suspicious strings, the number of DOM modification functions, the script's whitespace percentage, the average length of the strings used in the script, the average script line length, the number of strings containing "iframe," the number of strings containing the name of tags that can be used for malicious purposes, the length of the script in characters. Hereinafter, we provide some details about a subset of these features.

Keywords-to-words ratio. This feature represents the ratio between the number of keywords (i.e., reserved words) and other strings occurring in a piece of JavaScript code. This feature is useful to detect malicious pages because in most exploits the number of keywords (e.g., `var`, `for`, `while` and few others) is limited while there are usually a large number of other operations (such as instantiations, arithmetical operations, function calls). This usually does not happen in benign scripts, where the occurrence of keywords is usually higher.

Number of long strings. This feature counts the number of "long" strings used in this script. A string is considered long if its length is above a certain threshold. This threshold is learned during the training phase by examining the length of strings in both known benign and known malicious pages (40 characters in our experiments).

Presence of decoding routines. This feature expresses whether the JavaScript script contains snippets of code that resemble decoding routines. More precisely the AST of the JavaScript segment is analyzed to identify loops in which a "long" string is used (where "long" is defined according to the feature described before). This feature is very effective in detecting routines used to decode obfuscated scripts.

Shellcode presence probability. This number expresses the probability that a JavaScript script contains shellcode. We analyze the long strings contained in the script to check if their structure resembles shellcode. We use three methods to determine if the string is likely to represent shellcode. The first method considers the number of non-printable ASCII characters in the string. The second one detects shellcode composed only of hexadecimal characters, i.e., it checks if the string is a consecutive block of characters in the ranges a-f, A-F, 0-9. The third method checks if certain characters are repeated at regular intervals in the string, because sometimes the bytes of the shellcode are concatenated using custom separators, so that decryption routines can split the string over the specified separator(s) for further processing. The final shellcode probability for a certain script is set to the maximum of the results produced by the three individual detection methods.

Number of direct string assignments. This feature counts the number of string assignments in the script. To extract this feature, we analyze the structure of the AST generated by the parser. We consider a number of ways in which a JavaScript program can instantiate a string. More precisely, we count string assignments done through direct assignment, setting of properties, direct string declaration, instantiations inside the conditional operator '?', and arrays. The rationale behind this feature is that malicious scripts tend to have an unusually large number of string assignments, as a side effect of deobfuscation and decryption procedures.

Number of DOM-modifying functions. This feature counts the number of functions used to modify the Document Object Model that are referenced in the source code. Drive-by-download exploits usually call several of these functions in order to instantiate vulnerable components and/or create elements in the page for the purpose of loading external scripts and exploit pages. We consider the most-commonly-used DOM functions implemented in all the major browsers, plus a small set of functions which are only available in Microsoft Internet Explorer's JavaScript engine, such as *clearAttributes()*, *insertAdjacentElement()*, and *replaceNode()*. Note that we perform some limited static (data flow) analysis to identify cases where basic DOM elements (e.g., the `document` variable) are assigned to other variables that are later modified.

Number of event attachments. This is the number of calls to functions used to set event handlers on certain actions. Not all events are interesting for us, as drive-by-download attacks usually need only to be triggered as the page loads or to disable error reporting in case something goes wrong. So, we only count event attachments related to these events: *onerror*, *onload*, *onbeforeunload*, *onunload*. The functions that can be used to attach an event handler are *addEventListener()*, *attachEvent()*, *dispatchEvent()*, and *fireEvent()*.

Number of suspicious object names. This feature represents the number of objects with a suspicious name. These objects are identified using the list of exploitable objects already used by HTML features (see Section 3.1.1). However, since most of the exploits dynamically insert objects and ActiveX controls into web pages using JavaScript, we have to check for these components also in the JavaScript code.

Number of suspicious strings. This feature has been added after manually analyzing several dozens of malicious scripts and noticing that most of them, if not obfuscated, tend to use certain strings as variable or function names. Thus, we check whether a script contains such tell-tale signs (common strings are, for example, "evil," "shell," "spray," and "crypt"), and we count how many occurrences of these strings are found.

Number of "iframe" strings. This feature counts how many strings containing "iframe" are present in a script. This feature is motivated by the fact that malicious scripts often inject several iframes into a web page, and, if the script is not obfuscated, it is possible to identify when the script modifies the DOM to inject an `iframe` element.

Number of suspicious tag strings. Similarly to the previous feature, this feature counts the number of times that certain tag names appear inside strings declared in JavaScript code. In fact, instead of injecting iframes, sometimes malicious scripts write other scripts or objects inside the page. This feature counts the appearance of `script`, `object`, `embed`, and `frame` inside JavaScript strings.

Each piece of JavaScript code is also characterized by a hash of the content (to avoid analyzing a previously-seen script) and a signature of the AST of the document. This is used to identify similar scripts that have been already analyzed and that have been found to be malicious. If a match is found, the web page is considered malicious, and it is sent to the dynamic analysis tool for further processing. To prevent simple obfuscation techniques from hiding the similarity with other scripts, this AST signature does not take into account variable names and the structure of arrays, and it is invariant to the place where functions are declared.

3.1.3 URL and host-based features

As shown by previous work [17], it is often possible to predict if a certain web page is malicious by looking only at its URL. Even though detecting drive-by-download pages using URL features is more complex than in the case of phishing pages or scam pages, some information contained in the URL and associated with the referenced host can be used to help in the detection of malicious web pages. For example, malware campaigns are often hosted on untrusted hosting providers, and the corresponding `whois` information reveals short registration time frames or sanitized (anonymized) registration information. Also, it is very common for malicious web pages to include content from sites with no DNS name, or hosted on domains with a certain TLD (e.g., `.cn`, `.ru`).

The collected features are syntactical (the domain name length, whether the original URL is relative, the presence of a suspicious domain name, the TLD of this URL, the presence of suspicious patterns, the length of the file name appearing in the URL, the presence of a suspicious file name, the absence of sub-domain, the presence of an IP address in the URL, the presence of port number, the absolute and relative length of this URL), DNS-based (resolved PTR record, whether the PTR record is equal to the A record for this IP, and, for each of the A, NS, MX records for this host: first IP address returned, number of corresponding IP addresses, TTL of the first IP address, Autonomous System number of the first IP), whois-based (registration date, update date, expiration date), and geoip-based (country code, region, time zone, netspeed). We use a total of 33 features derived from the analysis of URL and host information. Below, we discuss some details about a subset of the features.

Number of suspicious URL patterns. Analyzing the URLs of several pages launching drive-by-download exploits, we observed that many of them shared common names or recurring patterns in their paths (we speculate that this is an indication that different attacks are performed using the same exploit toolkits, e.g., MPack, Eleonore, and CrimePack). Some examples of these patterns are file names such as `swfNode.php` or `pdfNode.php`. Thus, we use this feature to count how many patterns from a list of known bad patterns appear in the URL. We derive known bad patterns from known exploit kits. We currently identify 10 different suspicious URL patterns.

Presence of a subdomain in URL. We noted that, frequently, malicious web pages refer to the domains serving malware without specifying a subdomain (e.g., `example.com` instead of `www.example.com`). This feature keeps track of whether a subdomain is present in the URL.

Presence of IP address in URL. Some web sites hosting malware are not associated with domain names but are addressed by their IPs instead. A common reason for this is that the malware is hosted on a victim machine on a public network that was compromised. This feature records if an IP address is present as the host part in the URL.

Value of the TTL for the DNS A record. This feature examines the Time To Live (TTL) of the DNS entry of the first IP address returned by the DNS A query for a host name. Shorter TTLs are usually associated with services that are likely to be moved to another IP address in the near future. This can be the case for DNS entries associated with malicious (fast-flux) hosts.

Value of the TTL for the DNS NS record. This feature examines the Time To Live of the first NS entry for the host name under analysis. This feature is useful for identifying malicious web pages because criminals often use different DNS records to redirect requests to a different IP address once one of their command-and-control servers is shut down.

Relationship between PTR and A records. This feature indicates whether the resolved PTR record equals the IP address for the host under examination. For benign web servers, the values should be consistent.

Registration date. This feature examines the registration date for the host name (domain), if it is available via the Whois service. Registration dates are commonly used to distinguish between benign and malicious domains, since most of the command-and-control and exploit servers reside on domains whose registration date is recent and/or whose expiration date is in the near future. This is because attackers often buy domain names for short time frames, since they expect that those names will be blocked quickly.

Country Code. This feature leverages the country code to which the IP address of the host belongs. This feature is extracted via a geoip query[1].

Unlike previous work [17], we do not consider the domain registrar as one of our features. Even though we extract and store this information, the models we derived during the training process did not identify the registrar as a relevant feature for determining if a web page is malicious or not.

3.2 Discussion

Models and classification. In *Prophiler*, a model is a set of procedures that evaluate a certain group of features. More precisely, the task of a model is to classify a set of feature values as either likely malicious or likely benign. A model can operate in training or detection mode. In training mode, a model learns the characteristics of features as found in sets of web pages that are known to be either malicious or benign. In detection mode, the established models are used to classify pages as either likely malicious or likely benign.

Using a *training* dataset, we derived a number of models to detect likely malicious web pages, based on the features described earlier. The model learning process is further explained in Section 5. After training, we evaluated the effectiveness of our models on a *validation* dataset. Once we were confident that the models were able to effectively classify malicious web pages, we deployed them as a filter for our dynamic analysis tool (Wepawet). This resulted in a tenfold increase in the amount of web pages that the system can process with a given set of resources.

Machine learning. As with all classification problems, our learning-based approach to the detection of malicious web pages faces several challenges [28]. Here, we discuss in particular the assumptions at the basis of our analysis and the techniques we used to ensure that these assumptions hold in our setting. First, we assume that the distribution of feature values for malicious examples is different from benign examples. To ensure this, we carefully selected the features that we use in our system on the basis of the manual analysis of a large number of attack pages. We note that individual feature values may appear in both malicious and benign pages (e.g., some benign pages are obfuscated, thus they would "trigger" features that capture obfuscation). However, it is unlikely that the combination of all the features we consider is similar in benign and malicious pages. A second assumption is that the datasets used for model training share the same feature distribution as the real-world data that is evaluated using the models. We address this issue by training our models with large datasets of recent malicious and benign pages, and by continuously evaluating the effectiveness of our filter in detecting web pages with respect to the results provided by (more costly) dynamic analysis tools. A final requirement is that the ground truth labels for the training datasets are correct. To this end, we use state-of-the-art tools and manual inspection to ensure that our labeled datasets are correct.

Evasion. The attentive reader will notice that some of our features could be evaded by malicious scripts. For example, the detection of tags with a small area (one of our HTML features) could be thwarted by dynamically generating these elements (e.g., via an obfuscated call to *eval()*). However, our set of features is comprehensive and covers characteristics that are common to malicious scripts (e.g., the use of obfuscated code). As a consequence, as our experiments show, our system is not easily evaded by current malicious web pages. Moreover, it is easy to extend *Prophiler* with additional features to cover future attacks. We always send to the back-end analysis (honeyclient) a small fraction of random pages that our system has classified as benign. This allows us to detect systemic false negatives, and to update our feature sets and models accordingly.

Even with full knowledge of our feature set, it is not trivial for an attacker to disguise his malicious code. First, in certain cases, the freedom of an attacker is limited with regard to the parts of the infected web page that he can modify. In particular, this is true when attackers use SQL injection vulnerabilities, which often result in out of place HTML elements that cannot be cleaned up (and which are picked up by our system). Second, many of our features do not target specifics of particular exploits, but general properties of entire classes of attacks. For example, artifacts that are the result of obfuscated JavaScript are hard to disguise. Of course, an attacker could opt to send the exploit code in the clear, but in doing so, he risks that signature-based solutions detect and block the malicious code.

Attackers could also try to fingerprint, detect, and consequently evade, our tool when it visits a malicious website. This is a problem every malware detection tool faces, and we address it in two ways. First, we configure our system so that it closely mimics a

[1] Geoip queries are used to retrieve location information about an IP address. Usually, this information includes the country, region, and city to which the IP address belongs, as well as some other information such as the Internet Service Provider of this address, depending on the geoip service in use.

real browser (for example, by setting the user-agent of the crawler component as described in Section 4). Second, we try to detect fingerprinting attempts by using features that check for the presence of JavaScript routines commonly used for this task (as discussed in Section 3.1).

Trade-offs. Even though we put great care in the selection of the features and the derivation of models, we do not expect our filter to be as accurate as honeyclients, which can rely on the dynamic characteristics of a web page for the detection of malicious behavior. Instead, we expect the filter to provide useful information that can be used to quickly discard benign web pages, and to send *likely* malicious pages to dynamic analysis tools, which can perform more detailed analysis.

In this context, it is critical to minimize false negatives, i.e., missed detections. In fact, if a page that is indeed malicious is incorrectly classified as benign by our filter, it will be immediately discarded without being further analyzed. Therefore, the malicious page will evade the detection of the combined filter/honeyclient system. Conversely, false positives are not as problematic: If a benign page is incorrectly flagged as likely malicious by our filter, it will simply be forwarded for analysis to the honeyclient, which (we assume) will ultimately mark it as benign. The net effect is that resources are wasted (because the back-end honeyclient has to analyze a benign page). However, in this case, no incorrect detection will be made by the overall detection system.

4. IMPLEMENTATION AND SETUP

We implemented *Prophiler*, and we used it as a filter for our existing dynamic analysis tool, called Wepawet [4] (which is publicly available at `http://wepawet.cs.ucsb.edu/`). However, *Prophiler* can be used unchanged as a filter for any of the other, publicly available honeyclient systems. The overall architecture of the system is shown in Figure 1.

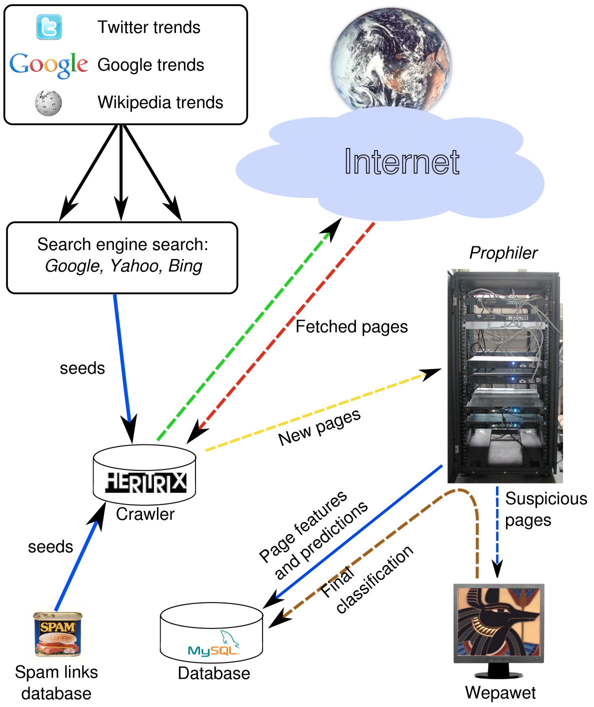

Figure 1: Architecture of the system.

Prophiler is fed by a modified instance of Heritrix [10], which crawls[2] a list of seed URLs fetched daily from three search engines (namely, Google, Yahoo, and Bing). The crawls are seeded by using the current Twitter, Google, and Wikipedia trends as search terms. These trends are used as a basis for the searches because most malware campaigns use Search Engine Optimization (SEO) techniques to increase their ranking in the search engines' results associated with popular terms [11, 12]. Another source of seeds for our crawler is a list of links extracted from a feed of spam emails. The list of links is updated daily and provides us with an average of 2,000 URLs per day.

We modified the crawler to be able to set the "Referer" header when fetching a seed URL. This header has to be set to the search engine from which the seed URL was extracted. This is necessary because some malicious web pages deliver malicious content only when the request appears to be the result of a user clicking on the search results.

The crawler fetches pages and submits them as input to *Prophiler*, which analyzes each page and extracts and stores all the features. Once all features have been extracted from a page, *Prophiler* uses the models learned in the previous training phase to evaluate its maliciousness. If a page has been identified as likely malicious, it is forwarded to the dynamic analysis tool (Wepawet, in our case). This tool then confirms that the page is indeed malicious or it flags it as a false positive.

The system was installed on a server running Ubuntu Linux x64 v 9.10, with an 8-core Intel Xeon processor and 8 GB of RAM. The crawler and the analysis system are both running on this machine. The system in this configuration is able to analyze on average 320,000 pages/day. Taking into account that a single page can contain multiple links to JavaScript programs, frames, and objects (which are all automatically included by the browser when rendering the page), the analysis must examine around 2 million URLs (objects) each day.

5. EVALUATION

In this section, we evaluate the effectiveness and performance of *Prophiler*. More precisely, we first discuss how the models used to detect malicious pages were automatically derived from a training dataset. Then, we evaluate *Prophiler* on a number of datasets, both labeled and unlabeled, adding up to almost 20 million web pages. Finally, we quantitatively compare our approach with those that were proposed in the past.

Model derivation. To derive our detection models, we collected a labeled dataset composed of both malicious and benign pages. We refer to this dataset as the *training dataset*. As shown in Table 2, the training dataset comprises 787 pages that are known to be triggering drive-by-download attacks. These pages were extracted from Wepawet's database. Furthermore, we confirmed by manual inspection that these pages indeed contain malicious code used to launch drive-by-download attacks. We also collected a set of 51,171 benign web pages by crawling the first two levels of the top 100 Alexa web sites [1]. In this case, our assumption was that these extremely popular web sites are unlikely to have been compromised and used for malware distribution, as they are visited daily by millions of users, as well as continuously analyzed by experts and antivirus programs. Furthermore, we used the Google Safe Browsing API to remove any malicious pages from this set.

[2] Most drive-by-download attacks use browser fingerprinting to decide whether to 'render themselves' as malicious or benign. We decided to set up our crawler's user-agent as *MS Internet Explorer 6* on Windows XP, to trigger malicious behavior in most cases.

Dataset name	Benign pages	Malicious pages	Total pages
Training	51,171	787	51,958
Validation	139,321	13,794	153,115
Evaluation	N/A	N/A	18,939,908
Comparison	9,139	5,861	15,000

Table 2: Datasets used for our experiments.

We extracted our detection models from this dataset using the WEKA machine-learning platform [9]. We experimented with a number of standard models, such as naïve Bayes, random forest, decision tree, and logistic regression classifiers. In order to choose a suitable classifier (i.e., the one providing the lowest possible number of false negatives, and a reasonably small amount of false positives) we used our training dataset to build several models, each with a different classifier and/or different parameters. The models were extracted from and tested on the training dataset, using 10-fold cross-validation.

Note that we built three different models that operate on the three different feature sets that we defined previously (HTML features, JavaScript features, and features related to the URL and host name). This allows us to evaluate the effectiveness of individual feature sets and to experiment with different machine learning models. For the final classification of a page, the output of the three models needs to be combined, as discussed below.

The results for the three classifiers (using 10-fold cross validation on the *training set*) are presented in Table 3. It can be seen that the classifiers that produced the best results were the Random Forest classifier for the HTML features, the J48 classifier for the JavaScript features, and the J48 classifier for the URL- and host-based features. In the rest of the experiments, we configured *Prophiler* to use these classifiers.

Feature class	Classifier	% FN	% FP
HTML	Random Tree	30.4	0.8
	Random Forest	20.5	2.4
	Naive Bayes	16.4	44.1
	Logistic	25.6	17.1
	J48	36.6	0.8
	Bayes Net	15.1	23.2
JavaScript	Random Tree	22.4	0.2
	Random Forest	18.1	0.5
	Naive Bayes	51.5	1.0
	Logistic	81.0	0.0
	J48	21.4	0.3
	Bayes Net	39.9	1.7
URL + HOST	Logistic	9.3	1.0
	J48	9.6	0.6

Table 3: False Negatives (FN) and False Positives (FP) ratios for the tested classifiers. The class of features related to the URL and host information has been tested against fewer classifiers because most of them do not support date attributes.

Interestingly, as shown in Table 3, it can be seen that a single class of features is not sufficient to reliably determine the maliciousness of web pages. In fact, individual models yield both high false positive and high false negative rates. For example, when analyzing JavaScript features alone, even J48 (one of the best performing models for this class) produces 21.4% false negatives (with 0.3% false positives). However, as we will show with a number of tests on various datasets, combining models for all the feature classes substantially improves the detection capability of our tool. In *Prophiler*, we declare a page as malicious when one or more of the individual classifiers declare a page as malicious. The rationale for this decision is that a page's maliciousness may be determined by causes (e.g., an iframe tag or an HTML-based redirect) that are modeled by only one class of features. Therefore, whenever the model associated with a class of features classifies a page as likely malicious, *Prophiler* raises an alert. As a result, by combining models, we can substantially reduce the false negatives of the filter by accepting a minor increase in false positives (which are much less problematic, as discussed previously).

Effectiveness of new features. In the next step, we inspected the models generated by WEKA to determine the importance of the new features that we added compared to previous work. We found that some of these features were particularly effective in the detection of web pages launching drive-by downloads. Regarding the JavaScript features, some of the most important new features are shellcode presence probability (which is at the first level in the decision tree of the chosen J48 classifier), the presence of decoding routines, the maximum string length, and the entropy of the scripts and of the strings declared in it. Several new features related to HTML content appear to be very effective in the detection of malicious or infected web pages. Such features are the number of included URLs, the number of elements containing suspicious content, the number of iframes, the number of elements with a small area, the whitespace percentage of the web page, the page length in characters, the presence of `meta` refresh tags and the percentage of scripts in the page. As for URL and host related information, the most effective novel features introduced by our work are the TLD of the URL, the absence of a subdomain in the URL, the TTL of the host's DNS A record, the presence of a suspicious domain name or file name, and the presence of a port number in the URL.

Validation. After the model derivation, we validated *Prophiler* by running it on a second labeled dataset, which we refer to as the *validation dataset*. This dataset contained 153,115 pages that were submitted to the Wepawet service by its users over a period of 15 days. We labeled each page with the result produced by Wepawet: in total, there were 139,321 benign pages and 13,794 malicious ones. On this dataset, *Prophiler* produced a false positive rate of 10.4% and a false negative rate of 0.54%. In other words, if used as a filter on this dataset, *Prophiler* would immediately discard 124,906 benign pages, thus saving valuable resources of the more costly (dynamic) analyzer.

Table 4 shows which models triggered the detection of malicious web pages when running the system on the *validation* dataset. One can see that most of the pages are considered malicious because of their HTML features, and secondly because of the JavaScript ones.

Large-scale evaluation. We performed a large-scale evaluation of *Prophiler* by running it over a 60-day period on a dataset containing 18,939,908 pages. This dataset (which we refer to as the *evaluation dataset*) was built by leveraging the infrastructure described in Section 4. All the pages in the evaluation dataset are unlabeled.

Prophiler flagged 14.3% of these pages as malicious, thus achieving an 85.7% reduction of the load on the back-end analyzer (in our setup, the Wepawet service). With the current implementation of Wepawet, this corresponds to a saving of over 400 days of processing. Figure 2 shows in more detail the analysis statistics for the 60-day analysis period. (The variation in the number of pages processed per day depends mainly on the number of URLs used as seeds, and the type and complexity of the visited pages, for example, the number of external resources fetched by each page.)

After Wepawet had analyzed all the pages that were marked as malicious by *Prophiler*, we could determine the false positive of our filter. We found that the false positive rate for this dataset was

Number of pages	Reason of suspiciousness
124,906	**None (classified as benign)**
14,520	HTML
9,593	JavaScript
1,268	Request URL
814	JavaScript + HTML
806	Request URL + HTML
467	Included URL(s)
189	Request URL + JavaScript
181	Included URL(s) + HTML
130	Request URL + JavaScript + HTML
119	Request URL + Included URL(s)
46	Request URL + Included URL(s) + JavaScript + HTML
28	Request URL + Included URL(s) + HTML
17	Request URL + Included URL(s) + JavaScript
16	Included URL(s) + JavaScript
15	Included URL(s) + JavaScript + HTML

Table 4: Results on the *validation* dataset.

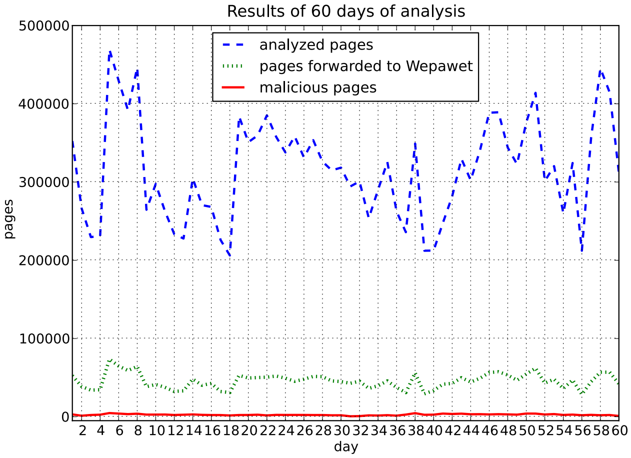

Figure 2: Analysis of the *evaluation* dataset. On average, 1,968 pages every day were confirmed as malicious by Wepawet.

13.7%. Recall that a false positive in our filter simply determines undesired load on the back-end analyzer (which is forced to inspect benign pages), but does not result in actual alert. Quantifying false negatives in these settings is more challenging, since the dataset is unlabeled and complete manual analysis is infeasible given the sheer size of the dataset. To estimate the false negative rate on the evaluation dataset, we processed with Wepawet 1% of the pages that *Prophiler* classified as benign (the pages to be further inspected were chosen at random). Of these 162,315 pages, only 3 were found to be malicious.

Comparison with previous work. We compared *Prophiler* against a number of previously-proposed systems that rely on lightweight analysis techniques to detect malicious web pages, and that, thus, could be used as fast (pre)filters.

More precisely, we considered the approach presented in [27], which relies on five HTML features and three JavaScript features to detect drive-by-download attacks; and the approach of [17], which analyzes URL features to detect malicious URLs. In addition, to better understand the effectiveness of the novel features that we introduced with respect to those that were proposed in the past, we created a classifier that combines all the features previously proposed in [5, 16, 17, 27]. We did not compare our system to [26] since obtaining four out of the six features they use (the ones related to redirects) was not possible for us; the other two features they propose are already included.

Unfortunately, we were also not able to compare our filter to the one used by Google [22]. The reason is that their filtering framework is proprietary (and not available to us) and is not publicly described in detail. Moreover, when using Google's Safe Browsing API, one is only able to check whether a page has been deemed malicious by Google's entire analysis framework, which is based on the use of honeyclients. However, no information can be retrieved about the false positive and false negative ratios of their initial filtering system.

To compare *Prophiler* with the above-mentioned systems, we built a labeled dataset (the *comparison dataset*) of 15,000 web pages and associated URLs. This dataset contains 5,861 pages involved in drive-by-download attacks; the remaining pages are benign. We modified our filter so that it would use only the features described in [27] (to reproduce the system described therein) and those presented in [5, 16, 17, 27] (to test the detection that can be achieved with all previously-known features combined). Finally, we asked the authors of [17] to analyze the URLs of the comparison dataset using their system.

Work	Features collection time	Classification time	FP %	FN %	Considered feature classes
[27]	0.15 s/page	0.034 s/page	13.70	14.69	HTML, JavaScript
[17]	3.56 s/URL	0.020 s/URL	14.83	8.79	URL,Host
Union of [5, 16, 17, 27]	N/A	N/A	17.09	2.84	HTML, JavaScript, URL, Host
Prophiler	3.06 s/page	0.237 s/page	9.88	0.77	HTML, JavaScript, URL, Host
Prophiler's top 3[*]	N/A	N/A	25.74	5.43	HTML, JavaScript, URL, Host
Prophiler's top 5[*]	N/A	N/A	5.46	4.13	HTML, JavaScript, URL, Host

Table 5: Comparison between *Prophiler* and previous work.
[*]These are, respectively, models built using only the top 3 and top 5 features appearing in the decision trees of *Prophiler*'s original machine learning models.

The results (in terms of average URL processing and testing time, false positives, and false negatives) are shown in Table 5. For the approach described in [17], we report the best results, which were achieved using an SVM classifier with an RBF kernel. *Prophiler* clearly outperforms existing approaches in terms of detection rates. In particular, it is interesting to observe that *Prophiler* has lower false positives and false negatives than the system that combines the features of [5, 16, 17, 27], indicating that the novel features and the models we use are effective and improve detection compared to the state of the art. Finally, the experiment also shows that *Prophiler*'s feature collection time is very low, despite the fact that it extracts a larger number of features than the other approaches. By profiling our tool, we found that the most of the feature collection time is due to the extraction of host-based features from the URLs (such as DNS information, Whois data, and geoip information). Note that the value of *Prophiler*'s features collection time refers to a "from scratch" run of the system, i.e., with an empty database. However, we found that a few hours after deployment, the database contains information about the majority of the hosts analyzed. Therefore, in steady state, our system is considerably faster, reaching a processing time of about 0.27 s/page.

6. CONCLUSIONS

As malware on the Internet spreads and becomes more sophisticated, anti-malware techniques need to be improved in order to be able to identify new threats in an efficient, and, most important, automatic way. We developed *Prophiler*, a system whose aim is to provide a filter that can reduce the number of web pages that need to be analyzed dynamically to identify malicious web pages. We have deployed our system as a front-end for Wepawet, a well-known, publicly-available dynamic analysis tool for web malware. The results show that *Prophiler* is able to dramatically reduce the load of the Wepawet system with a very small false negative rate.

Acknowledgements. The research leading to these results was partially funded from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n°257007 and Secure Business Austria. This work has also been supported by the National Science Foundation, under grants CNS-0845559 and CNS-0905537, and by the ONR under grant N000140911042.

We would like to thank Justin Ma for analyzing our *comparison dataset* and providing us with the results of the system he is author of.

7. REFERENCES

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

[2] Clam AntiVirus. http://www.clamav.net/, 2010.

[3] A. Clark and M. Guillemot. CyberNeko HTML Parser. http://nekohtml.sourceforge.net/.

[4] M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of the International World Wide Web Conference (WWW)*, 2010.

[5] B. Feinstein and D. Peck. Caffeine Monkey: Automated Collection, Detection and Analysis of Malicious JavaScript. In *Proceedings of the Black Hat Security Conference*, 2007.

[6] S. Garera, N. Provos, M. Chew, and A. D. Rubin. A Framework for Detection and Measurement of Phishing Attacks. In *Proceedings of the Workshop on Rapid Malcode (WORM)*, 2007.

[7] D. Goodin. SQL injection taints BusinessWeek.com. http://www.theregister.co.uk/2008/09/16/businessweek_hacked/, September 2008.

[8] D. Goodin. Potent malware link infects almost 300,000 webpages. http://www.theregister.co.uk/2009/12/10/mass_web_attack/, December 2010.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11(1):10–18.

[10] Heritrix. http://crawler.archive.org/.

[11] M. Hines. Malware SEO: Gaming Google Trends and Big Bird. http://securitywatch.eweek.com/seo/malware_seo_gaming_google_trends_and_big_bird.html, November 2009.

[12] W. Hobson. Cyber-criminals use SEO on topical trends. http://www.vertical-leap.co.uk/news/cybercriminals-use-seo-on-topical-trends/, February 2010.

[13] HoneyClient Project Team. HoneyClient. http://www.honeyclient.org/, 2010.

[14] A. Ikinci, T. Holz, and F. Freiling. Monkey-Spider: Detecting Malicious Websites with Low-Interaction Honeyclients. In *Proceedings of Sicherheit, Schutz und Zuverlässigkeit*, 2008.

[15] JSUnpack. http://jsunpack.jeek.org, 2010.

[16] P. Likarish, E. Jung, and I. Jo. Obfuscated Malicious Javascript Detection using Classification Techniques. In *Proceedings of the Conference on Malicious and Unwanted Software (Malware)*, 2009.

[17] J. Ma, L. Saul, S. Savage, and G. Voelker. Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2009.

[18] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A Crawler-based Study of Spyware in the Web. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2006.

[19] Mozilla Foundation. Rhino: JavaScript for Java. http://www.mozilla.org/rhino/.

[20] J. Nazario. PhoneyC: A Virtual Client Honeypot. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[21] D. Oswald. HTMLParser. http://htmlparser.sourceforge.net/.

[22] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFrames Point to Us. In *Proceedings of the USENIX Security Symposium*, 2008.

[23] P. Ratanaworabhan, B. Livshits, B., and Zorn. Nozzle: a defense against heap-spraying code injection attacks. In *Proceedings of the USENIX Security Symposium*, 2009.

[24] K. Rieck, T. Krueger, and A. Dewald. CUJO: Efficient Detection and Prevention of Drive-by-Download Attacks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2010.

[25] C. Seifert and R. Steenson. Capture-HPC. https://projects.honeynet.org/capture-hpc, 2008.

[26] C. Seifert, I. Welch, and P. Komisarczuk. Identification of Malicious Web Pages Through Analysis of Underlying DNS and Web Server Relationships. In *Proceedings of the LCN Workshop on Network Security (WNS)*, 2008.

[27] C. Seifert, I. Welch, and P. Komisarczuk. Identification of Malicious Web Pages with Static Heuristics. In *Proceedings of the Australasian Telecommunication Networks and Applications Conference (ATNAC)*, 2008.

[28] R. Sommer and V. Paxson. Outside the Closed World: On Using Machine Learning For Network Intrusion Detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[29] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2009.

[30] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2006.