DETECTING PDF JAVASCRIPT MALWARE USING CLONE DETECTION

by

Saruhan A. Karademir

A thesis submitted to the Department of Electrical and Computer Engineering

In conformity with the requirements for

the degree of Master of Applied Science

Queen's University

Kingston, Ontario, Canada

(September, 2013)

Abstract

One common vector of malware is JavaScript in Adobe Acrobat (PDF) files. In this thesis, we investigate using near-miss clone detectors to find this malware. We start by collecting a set of PDF files containing JavaScript malware and a set with clean JavaScript from the VirusTotal repository. We use the NiCad clone detector to find the classes of clones in a small subset of the malicious PDF files. We evaluate how clone classes can be used to find similar malicious files in the rest of the malicious collection while avoiding files in the benign collection. Our results show that a 10% subset training set produced 75% detection of previously known malware with 0% false positives. We also used the NiCad as a pattern matcher for reflexive calls common in JavaScript malware. Our results show a 57% detection of malicious collection with no false positives. When the two experiments' results are combined, the total coverage of malware rises to 85% and maintains 100% precision. The results are heavily affected by the third-party PDF to JavaScript extractor used. When only successfully extracted PDFs are considered, recall increases to 99% and precision remains at 100%.

Co-Authorship

The research described in Chapter 4 will appear in the *IBM Centre for Advanced Studies Conference (CASCON 2013*) [Kar13]. The paper was co-authored with my supervisors Sylvain P. Leblanc and Thomas R. Dean. I am the primary author.

Table of Contents

List of Figures

List of Tables

Glossary

Clone Class – The equivalence class formed by all pairs of source code segments that are considered similar by a clone detector.

Malicious Training Set – A set of malicious PDF files used to find an initial set of clone classes that will be used to find malware in the malicious evaluation set.

Malicious Evaluation Set – A set of malicious PDF files used to evaluate the effectiveness of clone detection.

Benign Evaluation Set- A set of PDF files containing JavaScript that does not contain malware. Used as a control group.

Signature Set – a minimal subset of the malicious training set that contains one file from each clone class in the training Set and all singleton elements in the training set.

Chapter 1

Introduction

1.1 Motivation

The field of computer security is a rapidly changing landscape dictated by the never-ending cat-and-mouse game played by malware developers and security analysts. As traditional targets, such as operating systems and web browsers, have been hardened, malware developers have turned their attention to third-party systems, such as Adobe PDF readers [Ado06,Ecm99]. PDF malware has skyrocketed in popularity since 2009 and has remained a high value target for attackers [Nvd13]. PDF's ubiquitous nature was a factor in its exploitation during a recent attack on industrial contractors involved in the drone manufacturing [Hon13]. These types of attacks exploit PDF's feature set such as embedded JavaScript [Ado06] to arbitrarily execute code on victims' systems.

Along with traditional client-side protection, intrusion detection systems (IDS), firewalls, and proxy servers are additional lines of defense against malicious PDFs [Roe99]. For these infrastructure-level security services, dynamic analysis of JavaScript is not permissible, because executing JavaScript on those platforms is a security risk and a performance concern.

Scripting languages, including JavaScript, are not compiled but are distributed in their original source form. This unique feature makes them amenable to source code analysis

techniques that go beyond typical malware detection techniques on used for compiled code.

Clone detection is an active research area with applications ranging from source comprehension, software evolution analysis, repository mining, pattern detection, and plagiarism detection. Various tools for detecting clones within and between source files have been developed by researchers with varying degrees of efficacy [Roy09]. The NiCad clone detection tool has proven effective in finding near-miss clones in source code [Roy08].

Our hypothesis is that near-miss clone detection will provide a means of detecting mutations of known malicious scripts.

1.2 Research Aim and Contributions

The aim of this research is to determine the feasibility of using clone detection techniques in detecting script-based malware. In addition to meeting this stated research goal, the research also makes the following important contributions:

- Experiment evaluating the parameters of clone detection for the purposes of classifying JavaScript malware
- Investigation evaluating the use of clone detection as a pattern matching engine for malware detection
- Analysis of syntactic levels appropriate for detecting PDF JavaScript malware

1.3 Outline

Chapter 2 is an overview of the background information and related work in the field of cloning and malware detection. Chapter 3 describes initial experimentation with NiCad and the setup of the two experiments conducted. Chapter 4 presents the results and analysis of the clone signature detection experiment. Chapter 5 explains the reflexive call detection experiment and its results. Chapter 6 concludes the thesis with future avenues for research.

Chapter 2

Background

2.1 Introduction

This chapter discusses the foundational concepts of the research including source code cloning, the PDF document format along with its support of JavaScript, and the VirusTotal malware database. Section 2.2 discusses the definition of cloning in software development and Roy and Cordy's NiCad Clone Detector tool. Section 2.3 provides a brief overview of PDF document structure and its support for JavaScript. Section 2.4 is an introduction to PDF malware utilizing JavaScript. Finally, section 2.5 describes the VirusTotal online malware database for researchers.

2.2 Cloning and NiCad

This section is a discussion of source code cloning and the NiCad Clone Detection tool. Subsection 2.2.1 introduces the concept of cloning, details cloning categories, and provides a brief overview of the current state of the art in clone detection. The clone detection tool, NiCad, is described and its high-level architecture is explained in Subsection 2.2.2.

2.2.1 Clones

Cloning is the reuse of existing source in software development. It is a common practice in developing large-scale projects. Some academic research of clones has been in detecting clones within large singular projects or cross-examination of different iterations

of the same project [Roy09]. This scenario for clone detection aids developers' code comprehension and also ensures that when one instance of a clone is updated, other copies of the code are also examined. Clone detection can also be used to find code that has been illegally copied from other software [Lan04].

As malware detection software is updated with signatures of existing malware, the authors of the malware make changes to bypass the anti-malware detection. As stated in Chapter 1, our hypothesis is that clone detection can be used for to find these modified malware copies.

Source code clones are categorized into four types in order of decreasing similarity [Roy09].

Type 1 clones are exact replications of code. This type of clone is the easiest to detect using standard signature based approaches. It is however detectable by lexical malware detection methods.

Type 2 clones are also called renamed clones. Variable, literal, and/or type names differ from clone to clone but the code structure is otherwise identical. Aesthetic features such as formatting and comments, which are ignored by the compiler or interpreter, can also differ. The malware author may also have changed some of the variable values to change the IP address or domain name of a command and control server used to control the malware after infection. Type 2 clones are commonly found in malware as a very basic form of obfuscation.

Figure 2.1 presents an example of a malicious type 2 clone pair in JavaScript. While both code segments accomplish the exact same task of reflexively calling the `eval` function

(with malicious parameters), the variable names used in the code differ. We have not shown the actual malicious code assigned to the string variable `maliciousCodeInString` as it is formatted into a single string and too long for the space in the figure.

```
a = 'eval';
this[a](maliciousCodeInString);

b = 'eval';
this[b](maliciousCodeInString);
```

Figure 2.1 - Example Type 2 Clone Pair

Type 3 clones are called near-miss clones. These types of clones can have differences in addition to the renaming of variables and values. Additional lines of code can be added; existing lines of code can be removed, modified or moved. These changes are made by malware developers to further disguise their malware and add structural obfuscation. Figure 2.2 demonstrates a Type 3 clone of the code previously presented in Figure 2.1. While Figure 2.2 retains the original lines from Figure 2.1 there is also a new line that performs an additional task. The malicious code hidden in the string has been further obfuscated by replacing spaces with the `'x'` character (e.g. `"a=b"` → `"ax=xb"`). This extra line prepares the second stage of JavaScript for evaluation by reversing the obfuscation by replacing the `'x'` character with spaces. Near-miss clones do not have a universally accepted threshold of difference. For this reason, when approaching clone

detection, a certain tolerance for unique features must be set to prevent false positive detection. A 30% uniqueness threshold is the default value used by the NiCad clone detection suite. This means that 70% of the lines of the code must be identical, except for renaming, for two code segments to be considered clones of each other.

```
b = 'eval';
this[b](maliciousCodeInString);
```

```
b = 'eval';
maliciousCodeInString.replace('x',' ');
this[b](maliciousCodeInString);
```

Figure 2.2 - Example Type 3 Clone Pair

Type 4 clones are called semantic clones. A type 4 clone may not have any syntactic similarity to its counterparts but it performs the same exact functionality. NiCad does not detect these types of clones, and they are outside the scope of this research.

A *clone class* is the equivalence class formed by the transitive closure of the clone pairs. That is, if source code segments a and b are clones and source code segments b and c are clones, then the segments a, b, and c form a clone class.

2.2.2 NiCad (Automated Detection of Near-Miss Intentional Clones)

NiCad is a parser-based clone detection tool with a plug-in architecture [Roy08]. NiCad's architecture is comprised of three stages as shown in Figure 2.3. The first two stages are language-specific; new languages can be supported using plug-ins. NiCad plug-ins come

in the form of TXL [Cor06] scripts that add support for more languages or configuration scripts that change the parameters of the clone detection process. The final stage is language independent and determines clone pairs based on a preset similarity threshold.

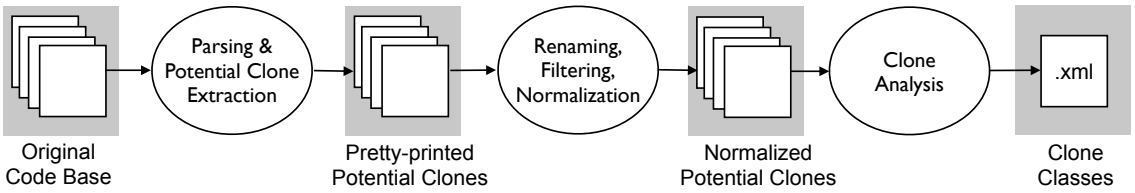


Figure 2.3 - NiCad Process (Adapted from [Cor11])

Stage One – Parsing and Extraction

The first stage of NiCad reads input source files and extracts potential clones using a user-specified syntactic boundary. This is a hard requirement for NiCad; no arbitrary sequence of characters can be considered as a potential clone. The fundamental restriction allows for faster but more guided detection. Extraction is accomplished through a TXL transformation specifically tailored for the target language and desired potential clone granularity. The choices for the syntactic boundary are limited by the TXL grammar for the target language. Typically, functions and blocks are used as the syntactic boundaries.

The extraction transform extracts every instance of the syntactic unit found in the source and encapsulates it in XML markup and saves it into an XML file. Figure 2.4 demonstrates an XML file built by NiCad when extracting `js_program` syntactic units from files in the `lib/folder`. Frequently, syntactic units will contain nested syntactic units of the same type. For example, a block may contain a nested block. In Figure 2.5,

the block in the dashed lines is extracted as one entity along with an additional extraction of the block surrounded in dotted lines. In these cases, both the parent and child unit will be extracted as potential clones. The XML markup also contains metadata such as file name and source coordinates.

```
<source file="lib/type1/subtype2/stage1.js" startline="1"
endline="19">
var z;
var y;
var h = 'edvoazcl';
z = y = app [h.replace (/[aviezjl]/g, '')];
var tmp = 'syncAEEotScan';
y = 0;
z [tmp.replace (/E/g, 'n')] ();
y = z;
var p = y.getAnnots ({nPage : 0});
var s = p [0];
s = s ['sub' + 'ject'];
var l = s.replace (/[zhyg]/g, '%');
s = unescape (l);
app [h.replace (/[czomdqs]/g, '')] (s);
s = '';
z = 1;
</source>
<source file="lib/type1/subtype3/stage1.js" startline="1"
endline="16">
var z;
var y;
z = y = app.doc;
y = 0;
z.syncAnnotScan ();
y = z;
var p = y.getAnnots ({nPage : 0});
var s = p [0].subject;
var l = s.replace (/z/g, 'a%b'.replace (/[ab]/g, ''));
var th = this;
s = th ['unes' + 'cape'] (l);
var e = app ['ev' + 'al'];
e (s);
</source>
```

Figure 2.4 - Example XML output from NiCad

Code.js

```
function ()
{
    for (var bkmrsw; otvwxy < delq.length; otvwxy + = 1) {
        var rzgcfp = otvwxy % fsvy.length + 1;
        var ansy = fsvy.substring (otvwxy % fsvy.length, rzgcfp);
        var acrv = delq [otvwxy];
        acdfm + = adgtuy (acrv - ansy.charCodeAt (0));
    }
    dfpr (acdfm);
};
```

example-block.xml

```
<source file="Code.js" startline="3" endline="9">
    for (var bkmrsw; otvwxy < delq.length; otvwxy + = 1) {
        var rzgcfp = otvwxy % fsvy.length + 1;
        var ansy = fsvy.substring (otvwxy % fsvy.length, rzgcfp);
        var acrv = delq [otvwxy];
        acdfm + = adgtuy (acrv - ansy.charCodeAt (0));
    }
    dfpr (acdfm);
</source>
<source file="Code.js" startline="4" endline="7">
        var rzgcfp = otvwxy % fsvy.length + 1;
        var ansy = fsvy.substring (otvwxy % fsvy.length, rzgcfp);
        var acrv = delq [otvwxy];
        acdfm + = adgtuy (acrv - ansy.charCodeAt (0));
</source>
```

Figure 2.5 - Example of nested blocks in JavaScript

By default, the extraction stage will pretty-print the potential clone. This pretty-printing can be modified to break single lines into multiple lines in order to increase the resolution of the clone analysis done in stage three. This XML encapsulated potential clone is the basic unit of NiCad that is used in all future stages of normalization, comparison, and presentation.

Stage Two – Normalization

Once extraction is complete, the potential clones can be immediately analyzed for similarity. To expand the detection capabilities of the system, NiCad provides three major normalization functions: renaming, filtering, and abstraction.

Renaming replaces all identifiers (as described in the language grammar) in a potential clone with a single identifier, defaulting to the character `x`. If desired, this type of renaming can be augmented with sequential naming by appending a number to the character `x` and maintaining that name for that specific variable, thus offering consistent renaming. In Figure 2.6 the variable is renamed as `x1` in line 1 and then, when called again in line three, it keeps its assigned `x1` name. NiCad supports both consistent renaming and blind renaming allowing it to detect type 2 clones.

No renaming

```
var num = 1;
pr = app.doc.getAnnots ({nPage : 0});
sum = pr [num].subject;
```

Blind renaming

```
var x = 1;
x = x.x.x ({x : 0});
x = x [x].x;
```

Consistent renaming

```
var x1 = 1;
x2 = x3.x4.x5 ({x6 : 0});
x7 = x2 [x1].x8;
```

Figure 2.6 - Comparison of renaming techniques

Filtering is an additional normalization technique that removes designated syntactic elements from the potential clones. For example, variable declarations can be removed from clone candidates to evaluate clone similarity based only on the placement and features of the leftover control and assignment statements. Figure 2.7 demonstrates the removal of the variable declaration in the first line of the `montRevert` function's block. Filtering can identify more clones by removing elements and reducing the impact of length on uniqueness of clone comparisons.

No Normalization

```
function montRevert (x)
{
    var r = nbi ();
    x.copyTo (r);
    this.reduce (r);
    return r;
}
```

Filtering of Variable Declarations

```
function montRevert (x)
{
    x.copyTo (r);
    this.reduce (r);
    return r;
}
```

Abstraction of expressions

```
function montRevert (x)
{
    var r = nbi ();
    js_expn;
    js_expn;
    return js_expn;
}
```

Figure 2.7 – Comparison of Filtering and Abstraction

Abstraction is similar to filtering in targeting a specific syntactic element for modification, but instead of removing the element outright as in filtering, it is replaced by its syntactic name. For example, all expressions in a potential clone can be abstracted and replaced by the text string `'js_expn'`. Figure 2.8 demonstrates the abstraction of all expressions in the `montRevert` function. The expression `'x.copyTo (r)'` is replaced by the non-terminal `'js_expn'`.

Stage Three – Clone Analysis

After normalization, the XML encapsulated clone candidates are compared against each other using the Longest Common Substring (LCS) algorithm [Ber00], similar to that used in the Unix diff program. The comparison determines a similarity between two clone candidates based on how many identical lines they share by ratio. By default NiCad assumes any pairs that share 70% or higher lines of code are clones.

NiCad Output

NiCad has two modes of clone detection: standard and cross clone. The standard mode gives NiCad a single source folder to examine all source files inside this folder are examined for clones. The second cross clone mode compares two separate folders of source code to find code clone pairs between the two systems; no clones are detected within the single folders in this mode.

For both modes, NiCad outputs clone detection results as an XML file as shown in Figure 2.8. Each entry in the XML file is a clone pair that contains the metadata (filename, starting line, ending line) for both elements of the pair and a similarity index in

percentage. The report only contains pairs that met or exceed the similarity threshold. If specified, NiCad can also build an XML report that clusters all similar clone pairs into clone classes as shown in Figure 2.9. Other aesthetic niceties include HTML-based reports and source-embedded reports.

```
<clones>
<systeminfo processor="nicad3" system="set"
granularity="fncall-blind-filter" threshold="30%"
minlines="1" maxlines="2500"/>
<cloneinfo npcs="64007" npairs="13"/>
<runinfo ncompares="2048288010" cputime="4378"/>

<clone nlines="1" similarity="100">
<source
file="set/114c4b379c26b55d5a2f70f177b82f41d7f79cfa28a949c
7d2e84c06f2eac10a.js" startline="1" endline="1"
pcid="59"></source>
<source file="libfncall/maliciouscalls.js" startline="3"
endline="3" pcid="64006"></source>
</clone>

<clone nlines="1" similarity="100">
<source
file="set/114c4b379c26b55d5a2f70f177b82f41d7f79cfa28a949c
7d2e84c06f2eac10a.js" startline="1" endline="1"
pcid="67"></source>
<source file="libfncall/maliciouscalls.js" startline="1"
endline="1" pcid="64004"></source>
</clone>
```

Figure 2.8 - Sample NiCad cross-clone detection XML report

```
<clones>
<systeminfo processor="nicad3" system="set"
granularity="fncall-blind-filter" threshold="30%"
minlines="1" maxlines="2500"/>
<cloneinfo npcs="64007" npairs="13"/>
<runinfo ncompares="2048288010" cputime="4378"/>
<classinfo nclasses="2"/>

<class classid="1" nclones="7" nlines="1"
similarity="100">
<source
file="set/114c4b379c26b55d5a2f70f177b82f41d7f79cfa28a949c
7d2e84c06f2eac10a.js" startline="1" endline="1"
pcid="59"></source>
<source file="libfncall/maliciouscalls.js" startline="3"
endline="3" pcid="64006"></source>
<source
file="set/25ae0ca0012008b46794ffd99a55f2e0b9683450f507f9e
a071634d5cc1e4135.js" startline="9" endline="9"
pcid="112"></source>
<source
file="set/33afd4969ec981d964d309d8bf76edd82a411590d9b9377
375f4471b60c2581b.js" startline="18" endline="18"
pcid="24254"></source>

...

</class>

<class classid="2" nclones="8" nlines="1"
similarity="100">
<source
file="set/7cd5b582665f9acb5f8fb9e836ba53f69f2db59b1dddd1a
8694122740eb85a95.js" startline="9" endline="9"
```

Figure 2.9 - Example NiCad cross-clone classes Report

NiCad also produces intermediate XML files that contain each source folder's XML encapsulated potential clones. Further intermediate XML files exist if any normalization is executed.

2.3 PDF and JavaScript

This section discusses the PDF document format and the use of JavaScript in PDF. Subsection 2.3.1 provides a structural overview of the PDF document standard and its constituent elements. Subsection 2.3.2 provides details on how JavaScript is stored in PDF files for execution by a PDF reader.

2.3.1 PDF file anatomy

Figure 2.10 is a visualization of a PDF document's internal layout [Ado06]. PDF files are divided into four sections: header, body, cross-reference table, and trailer. The body contains a sequence of objects. These objects can be of 8 types.

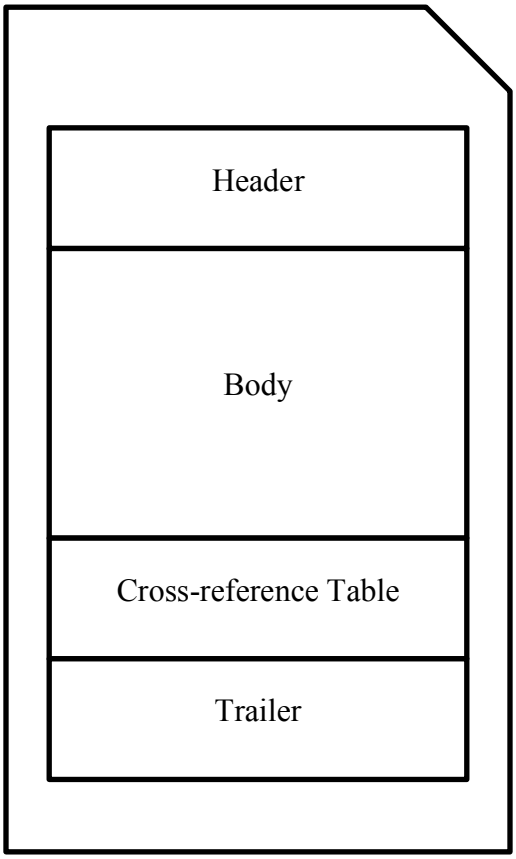


Figure 2.10 - PDF Document Layout - Adapted from [Ado06]

Five object types are scalar types: booleans, strings, real numbers, integers, and names. While most of these literal types match their procedural programming language counterparts, the `name` type is unique a case for PDF. Names play the role of symbolic constants and are identifiers preceded by a `/` character (e.g. `/Pages`). They are often used as keys and constant values in dictionaries. Strings in PDF are enclosed by the characters '`(`' and '`)`' or '`<`' and '`>`' (e.g. `(this is a string)`, `<deadbeef>`). The other three PDF object types are container types: array, dictionary, and stream.

Arrays are a list of objects enclosed by the tokens '`[`' and '`]`' and separated by white space.

Dictionaries are collections of name-object pairs (named dictionary entries) that are enclosed by the '<<' and '>>' tokens. Dictionaries have similar characteristics to a Lookup table in most languages.

Objects allow a single entity to be defined in one location and referenced elsewhere in the PDF document, possibly multiple times. Figure 2.11 shows and example object definition that also includes an object reference to an off-screen object with ID 5 Version 0.

The numbers leading the object definition are the object's identifier and version. In most cases and this example, the object is a dictionary. The dictionary has three entries. The first entry (`/Type /Pages`) acts as a header so the PDF parser can interpret the actual contents of the dictionary. In this example, this object is the pages dictionary. The second entry to the dictionary (`/Kids [5 0 R]`) is a mandatory field for the pre-defined `Pages` type. Kids is an array that represents the pages of the document. This Kids entry has one element in its array (`5 0 R`) which is a reference, as indicated by the trailing 'R', to an object with ID `5` version `0`. The third and final entry is a count of how many elements are in the /Kids array, which is one.

```
1 0 obj  <<
    /Type /Pages
    /Kids [ 5 0 R ]
    /Count 1
 >>
endobj
```

Figure 2.11 - Sample PDF Object Definition

Stream objects represent a stream of data that may be character or binary data. This may be anything from picture data (i.e. jpeg), to a common header or footer of a document or JavaScript. They follow the same general format as a regular object definition, but the object definition contains a dictionary and the stream data enclosed by the keywords `stream` and `endstream`. The dictionary uses a predefined set of keys to describe the data contained in the stream. Figure 2.12 shows an example of a stream definition.

At a minimum, the dictionary must contain a `/Length` entry which gives the length of the unencoded stream. The length entry is largely ignored by modern PDF readers, but it is still required. There are several optional stream dictionary entries, but one important entry is the `/Filter` entry. The filter entry lists a single name or an array of names that describes filters that are used to decode the data in the stream. For example the data may have been compressed using a particular compression library. Each of the filters must be used in the given order in to read the data. For example, an item that has a pair `/Filter [/ASCIIHexDecode /FlateDecode]` represents data that is first zlib compressed

and then ASCII hex encoded. Therefore, in orderorder to recover the data, the reader must first decode the ASCIIASCII hex encoding, and then expand the zlib compression.

```
30 0 obj <<
      /Filter /FlateDecode
      /Length 89
>>
stream
...
endstream
endobj
```

Figure 2.12 - Sample PDF Stream Definition

2.3.2 JavaScript in PDFs

Executable JavaScript[Ado06,Ecm99] can be found in entries of specific dictionaries in PDF documents as show in Figure 2.13. The script itself can either be embedded into the dictionary as a string value or contained in a stream referenced by a dictionary entry. The `/OpenAction` tag in a document's catalog dictionary is paired with a reference to an action dictionary. When a PDF reader encounters an `/OpenAction` tag, it executes the action linked to in the dictionary. In this example, the `/OpenAction` is paired with a reference to an action entry (`999 0 R`). Object 999 version 0 is a dictionary with the type action (`/Type /Action`). The `/S` entry defines the action as JavaScript and the `/JS` pair identifies the actual JavaScript code to be opened, in this case a stream object of id 777 version 0. Stream object 777 is compressed and ostensibly of unencoded length 11. It

should be noted that this `/Length` argument is intentionally incorrect to defeat some PDF decoding tools.

```
7 0 obj <<
  /Metadata 4 0 R
  /AcroForm 8 0 R
  /Pages 3 0 R
  /Type/Catalog
  /OpenAction 999 0 R
>>
endobj
999 0 obj <<
    /Type /Action
    /S /JavaScript
    /JS 777 0 R
 >>
endobj
777 0 obj <<
  /Filter /FlateDecode
  /Length 11
>>
stream
... compressed JavaScript ...
endstream
endobj
```

Figure 2.13 - Example JavaScript in PDF

2.3.3 Extracting JavaScript from PDFs – libPDFjs and Pita-J

Extraction of JavaScript from PDF documents is not a trivial task. While the PDF standard provides an explicit definition, most PDF readers, including the official Adobe Reader, are liberal in their acceptance of malformed PDF documents. This is a consequence of evolution and proliferation of the PDF format over its 20-year lifespan. However, the generally available PDF readers do not provide a convenient way to extract any and all JavaScript streams from PDF documents. This action requires the use of a specialized PDF JavaScript extraction tool. There have been prior attempts at creating such tools with mixed results on modern PDF malware.

The best third party results were produced by the libPDFjs, open source JavaScript extraction tool developed by Laskov and Šrndić [Las11]. LibPDFjs utilizes the open source Poppler PDF rendering library [Pop13] to interact with input PDF files. Its maturity means that it performs very well in extracting benign and some malicious PDFs. However libPDFjs had issues with certain malicious PDF files; it also depends on older versions of certain libraries to function.

Pita-J is a tool previously built by members of the research group to extract the JavaScript from PDF files [Pit13]. This functionality of this tool is more limited than libPDFjs and it does not handle as many internal encodings. One factor in the extraction process is the occasional addition of executable machine code appended to the PDF document. This x86 binary code payload is usually read, modified, and executed by the

embedded malicious JavaScript in the file. Most PDF parsers fail to read files that contain such malicious payloads.

To interpret PDF documents, Pita-J utilizes a simple island parser [Deu99] that identifies all top-level object definitions in PDF files. The island parsing approach was used because it was found to be more resilient to accidental or purposeful errors in the PDF document. Top level objects containing binary streams are marked and their contents are decoded. Not all stream encodings are supported by Pita-J. For example, streams encoded in a lossy compression algorithm, such as JPEG streams accompanied by the `/DCTDecode` tag, are ignored as they could not be used to store JavaScript. Dictionary items are examined for the `/JS` name, while cross-reference table and formatting items are ignored. If a `/JS` name's associated pair is an object reference, the PDF file is traversed to find the JavaScript containing object. This object's content is then read and printed out as a JavaScript file for inspection.

2.4 PDF-JavaScript malware

As operating systems and browsers have become more resilient, but not immune, to malicious attack, malware developers have expanded to web-facing products, such as the Flash plug-in or the Java Virtual Machine as attack vectors into a system [Ado08][Ora13]. PDF malware has become an increasingly common vector for attackers. A precipitous rise in reported Adobe Reader vulnerabilities starting in 2009 (Figure 2.14) has made PDFs of critical security interest [Nvd13].

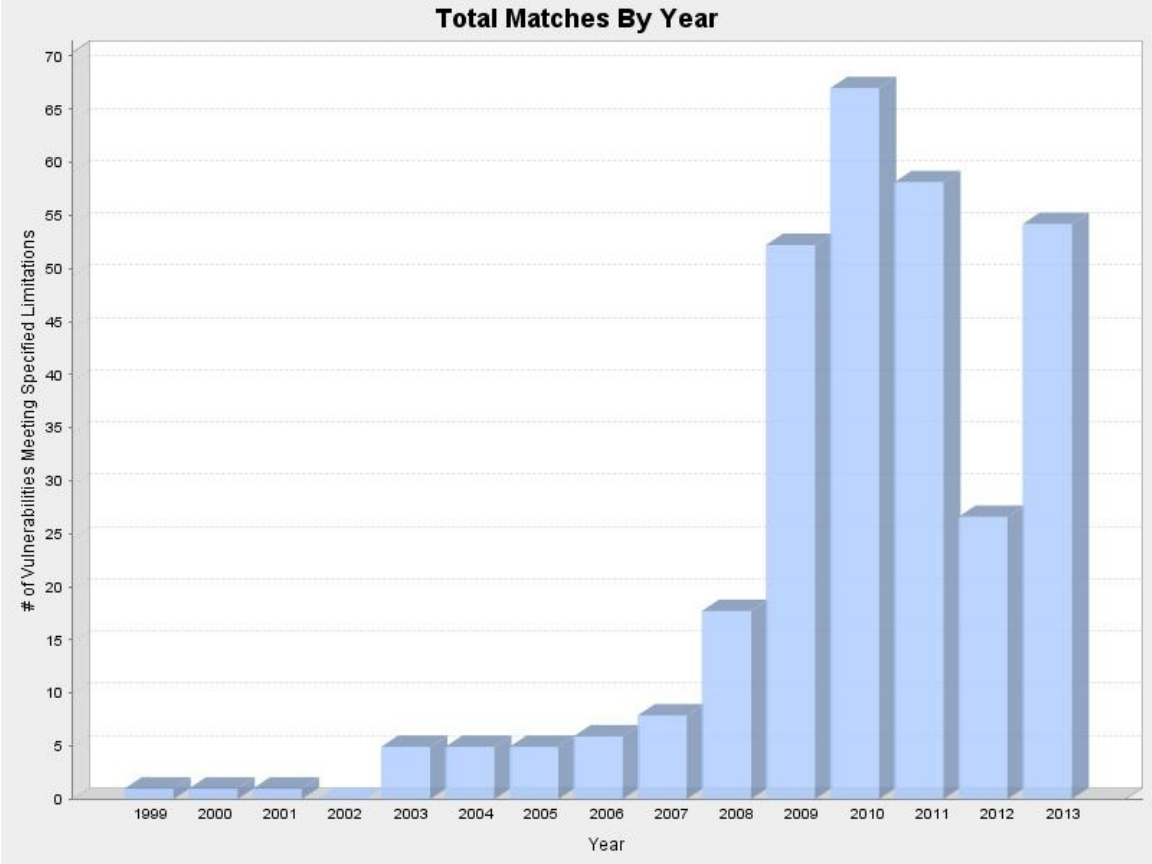


Figure 2.14 - Adobe Reader Vulnerabilities by Year [Nvd13]

PDF malware can be categorized as Trojan malware. Malicious PDF files do not propagate automatically, they are usually delivered to the user through embedded PDF in web pages or as attachments to emails. Most critical exploits of PDF readers are buffer overflows in PDF parsing, image rendering, or the JavaScript engine itself. The exploits themselves are usually initiated by embedded JavaScript in malicious PDF documents. Adobe and other PDF reader vendors have reacted to this rise in malicious PDF use by hardening their products and adding mitigation techniques when vulnerabilities are found. For example, Adobe Reader X added sandboxing protection that prevents

exploited readers from accessing memory or escalating privileges [Ado10]. In addition, Address Space Layout Randomization has been used to increase the difficulty of exploiting vulnerabilities in Adobe Reader.

2.4.1 Obfuscation Techniques

Malicious JavaScript in PDF files tend to be used to provide a delivery mechanism for more obvious and easy to detect malware. The payload may be in machine code or further JavaScript. This sort of code would look out of place upon manual inspection of the PDF file and easily detected by signature based malware detection. In an effort to bypass detection, malicious PDF files encode or obfuscate exploit code in different ways and in various locations in a PDF document. This obfuscated exploit code can be stored in PDF comment and metadata fields or in image fields to be later retrieved and executed by short JavaScript programs loaded by the PDF reader when the document opens. These small initial programs are called stage 1 unpackers. While not malicious in and of themselves, these unpackers are a strong indication of malicious intent because they serve no legitimate purpose. Authors have little reason to obfuscate benign JavaScript in this manner. PDF creation tools such as Adobe Acrobat provide encryption and compression features to reduce file size and protect assets. Unpackers can be nested within each other as well to cheaply avoid signature-based detection.

2.4.2 Unpackers

A sample stage 1 unpacker is provided in Figure 2.15. JavaScript unpackers perform three tasks. First, the encoded payload is retrieved. In this example, the second stage is

stored in an annotations field in the PDF document. PDF metadata is accessed by JavaScript using custom PDF-only functions, such as the app.doc.getAnnots({nPage:0}) call made on line 9.

```
var pr = null;
var fnc = 'ev';
var sum = '';

app.doc.syncAnnotScan();

if (app.plugIns.length != 0) {
	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(/-/);
	var proc = String.fromCharCode(22+15);
	for (var i = 1; i < arr.length; i++) {
		buf += String.fromCharCode("0x"+arr[i]);
	}
}

if (app.plugIns.length >= 2) {
	fnc += 'l';
	app[fnc](buf);
}
```

Figure 2.15 - Sample Stage 1 Unpacker

The second stage decodes the payload into its executable JavaScript form. In this example, the payload was series of hex character codes split by hyphens. The variable

‘buf’ is populated by converting the payload into its character values using the `String.fromCharCode("0x"+arr[i])` function. Finally, once the JavaScript string is in memory, it is executed using the `eval` function. The `eval` function in JavaScript accepts any string parameter and attempts to execute it as a JavaScript file within the same context as the caller. `Eval` functions are not used in benign JavaScript in PDF files and the keyword ‘`eval`’ is commonly used as a signature by anti-malware software. To evade the signature, malicious JavaScript files rarely explicitly use the ‘`eval(buf)`’ call, but instead obfuscate the call using a reflexive call. In this instance, the variable `fnc` is slowly built up to the string value ‘`eval`’ using various assignment statements as highlighted in Figure 2.15 on lines 2, 17 and 26 . The function call `app[fnc](buf)` is then executed (on line 27) which then performs the `eval` function on the second stage payload.

2.5 VirusTotal

VirusTotal is an Internet malware database.[Vir13] The public facing portion of VirusTotal allows any person to anonymously submit samples to the system for scanning. The submitted files are scanned using 42 to 47 different anti-virus solutions and their individual responses are posted online to be analyzed by both the submitter and researchers. The VirusTotal database itself is not directly accessible but researchers can access its information through a secondary portal called VirusTotal Intelligence. The search can be parameterized by various attributes of the files. Each file retrieved is

represented by a report with the results of VirusTotal's analysis and a link to download the original binary submission.

Along with the standard metadata of size, type, original filename, and submission dates, more malware specific information is encoded as tags. These tags include the number of positive detections from the suite of anti-malware software, the Common Vulnerability and Exposures (CVE) identifier, file format specific information such as embedded JavaScript or MySQL behavior of Windows Portable Executable submissions. Figure 2.16 provides a subset of total tags supported by the VirusTotal database. A researcher can build a specific search query from any number of these tags and receive a listing of all matching malware in the VirusTotal database. To download a collection of files, VirusTotal provides a batch download link on search results (Figure 2.17) and a python script for automation. These tools can be used by researchers to rapidly acquire a large set of test data.

Tag	*Description*
cve	The Common Vulnerability and Exposures identifier of the exploit that the file under consideration makes use of.
honeypot	The file was catched in the wild by a network honeypot setup, e.g. Dionaea honeypot.
spam-email	The file was seen as an attachment or download link in spam emails.
attachment	The file was seen as an attachment in some email, however, there is no certainty regarding whether such email was spam.
exploit	The file is or makes use of an exploit.
invalid-xref	PDF with an invalid xref table.
js-embedded	PDF that contains JavaScript.
flash-embedded	PDF that contains Flash.
autoaction	PDF that contains an automatic action to be performed when the document is viewed.
acroform	PDF that contains an AcroForm, which in turn may contain JavaScript that is executed when a document is opened.
launch-action	PDF that contains a launch action, which could launch a given JavaScript snippet.
file-embedded	PDF that contains an embedded file, could be executable code to launch via a launch action.

Figure 2.16 - Partial table of VirusTotal query tags [Vir13b]

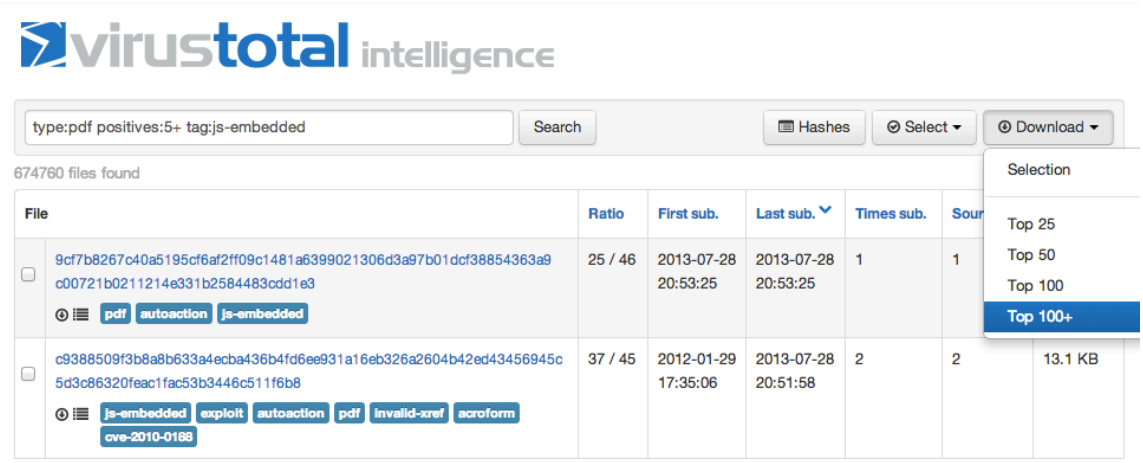


Figure 2.17 - VirusTotal Search Results page and Download options

VirusTotal has two major drawbacks. First, is the problem with volatility of results due to the temporal nature of VirusTotal's search system. Second, the inherent issue of trust in downloading benign files from a website dedicated to malicious file submission.

2.5.1 Volatility of results

While VirusTotal is a very powerful tool for researchers, its usage brings with it certain caveats. While it is easy to build a collection of malware circulating at the moment of search, it is harder to find historical examples of malware. VirusTotal's search results tend to favor the latest submissions. The search results page can be sorted by date first/last submitted, number of times submitted, size, and number of sources. However, the results will still be a subset of the submissions uploaded in the 30 days previous to the date of search. [Vir13b] A search parameter can be set to force a specific date, but that will not return any files that were last submitted more than a month ago. This limitation does not mean the collection of malware downloaded from VirusTotal is entirely recent. Many samples will be repeat submissions of files that have been submitted in the past. This limits the breadth of the VirusTotal-borne malware collection and makes it impossible consistently download the same malware. To alleviate this issue, the experiments carried out in this research project were conducted on very large sets of malware downloaded in one query rather than using multiple queries over time.

2.5.2 Confidence in Benign

Not all files submitted to VirusTotal are malicious and not all submitted malicious files are detected by the suite of anti-malware detectors. Novel approaches (such as zero-day

exploits) are rare in VirusTotal submissions, but many files that use known exploits are missed if the PDF files are password-protected. A search with the tag ‘positives=0’, intended to retrieve benign files may also contain undetected malicious files. To retrieve a genuinely benign set of PDF documents, one must manually examine the JavaScript in the downloaded files and VirusTotal’s information page (Figure 2.18) on the file. The VirusTotal information page contains a comments section that provides researchers the ability to comment on files and further explain how the files accomplish their malicious task; it can also note an undetected file is actually malicious. In practice, files rarely have comments, and only very frequent submissions gather the necessary attention for comments. In addition to manually added comments, the VirusTotal file information page gives the details of the automated analysis. This includes basic information for PDF documents such as the number of pages and the presence of JavaScript streams.

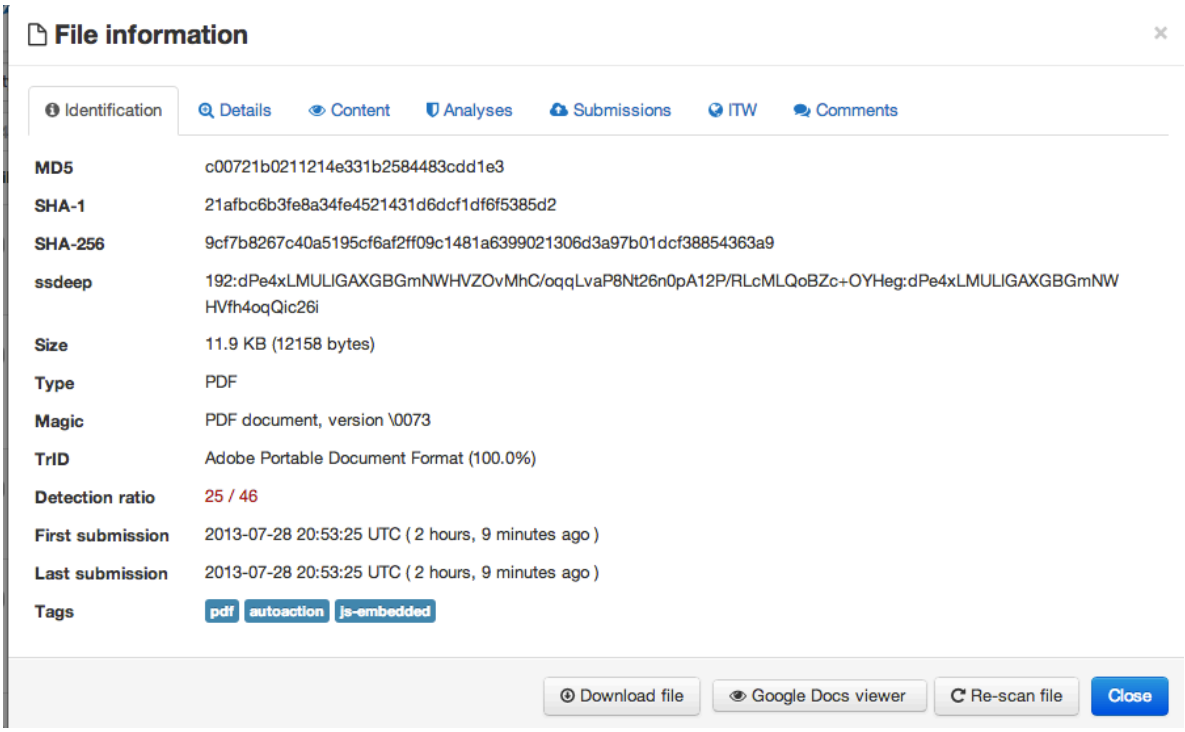


Figure 2.18 - VirusTotal File Information Pane

VirusTotal may mark a file as suspicious for containing both JavaScript and an automatic action, but it is up to the researcher to determine if this is malicious or benign behavior. Further complications arise as VirusTotal considers only the latest detection results to be valid. VirusTotal's library of anti-malware detectors is updated with new engines and signatures. Every time a file is resubmitted, it is reevaluated with this new set of detection tools, as seen in Figure 2.19. This can lead to files that were initially considered malicious being later classified as benign. There is no way for a researcher to filter such files from searches of benign files.

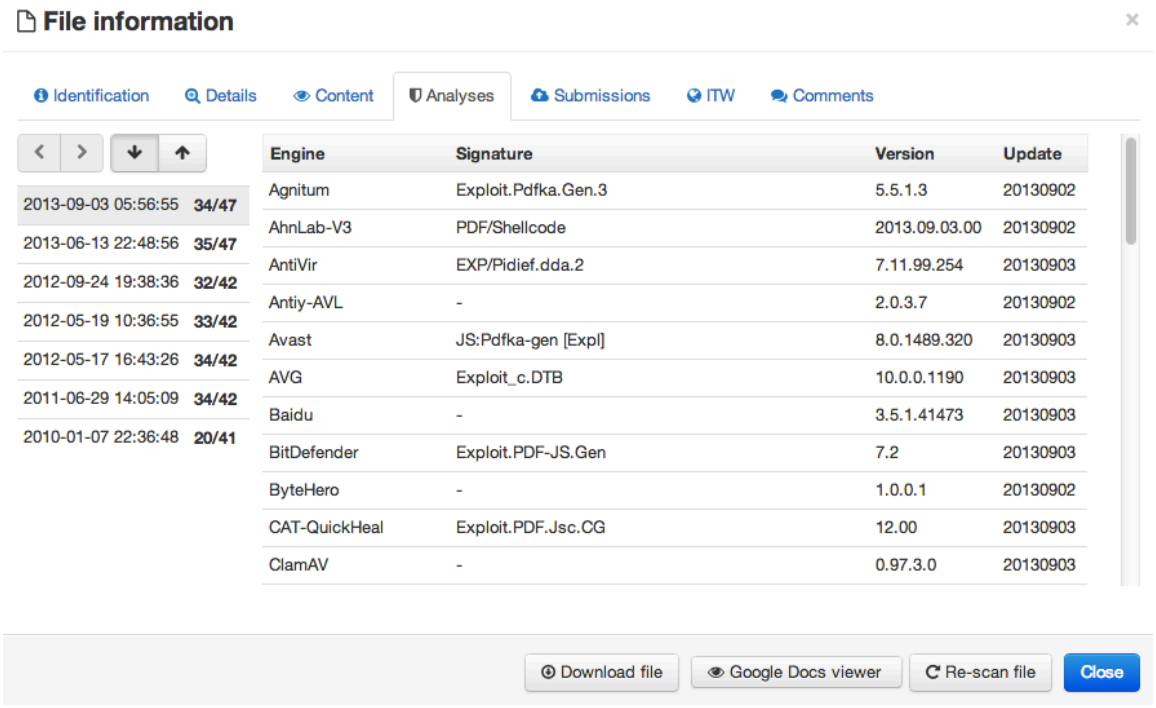


Figure 2.19 - VirusTotal's Repeat Analysis of files

2.6 Related Work

There have been various attempts at analyzing JavaScript's inherent properties to improve malware analysis beyond signature based detection. Some of these techniques target PDF-based JavaScript and others focus on HTML-embedded JavaScript.

2.6.1 PJScan

Laskov and Šrndić's [Las11] PJScan system provides a different method of classifying JavaScript malware in PDF documents. PJScan uses a token-based machine-learning algorithm in contrast to NiCad's hybrid parser/lexical approach to classification. A one-class support vector machine classification algorithm is trained using the stage 1 unpacker scripts extracted from 65,000 malicious PDF files from the VirusTotal

database. This trained classifier is then used to classify the JavaScript from an evaluation set of PDF documents. PJScan detects 85% of the malware present in the training set.. For new files that did not share any signatures with the training set, PJScan had a 71% detection rate. False positive rate on benign files was 16%. PJScan is the most similar tool to our research as it specifically targets PDF-borne JavaScript and only targets the stage 1 unpackers. PJScan's JavaScript extraction tool, libPDFjs, was used for the extraction of all benign PDF documents.

2.6.2 Zozzle

Microsoft Research's Zozzle [Cur11] is a browser-based JavaScript malware detection tool. It was purposed as both a real-time endpoint detection system for web browsers and also as an input filter for a more extensive diagnostic techniques. Zozzle uses a Bayesian classification system to determine the maliciousness of incoming files. Zozzle uses the abstract syntax tree (AST) from every JavaScript file and compares it to AST of known malicious and benign JavaScript exemplars. This system gives Zozzle a negligibly low false positive rate and very fast performance. Average throughput for Zozzle is roughly 1MB of JavaScript per second. Unlike our tool and PJScan, Zozzle is able examine code beyond the stage 1 unpacker. Zozzle hooks into Microsoft's `jscript.dll` JavaScript engine and executes the JavaScript source until it reaches an `eval` function call. At this point, the contents of the function call's parameter are passed to the classification system. This feature of Zozzle limits its application by requiring integration into the browser's

JavaScript engine. Web-based Javascript can be assembled from multiple online sources that dictate a more dynamic approach to detection.

2.6.3 Prophiler and Wepawet

Prophiler is a first pass filter applied to computationally-intensive dynamic analysis of malicious web pages [Can11]. It is uses static analysis and machine learning techniques to determine if a web page, along with its assorted HTML, JavaScript and even URL data, is benign. The tool does not analyze the JavaScript code itself. If Prophiler determines that a file is suspicious, it can pass it to an anti-malware tool that does actual JavaScript analysis to determine if the suspicion is justified. For this purpose Prophiler uses the Wepawet JavaScript tool by Cova et al [Wep13]. Wepawet uses a variety of means to determine malicious JavaScript code including a machine-learning technique emulating JavaScript code's behavior and then comparing the behavior to known execution profiles. Wepawet includes dynamic analysis techniques for other data types including Flash and PDF.

2.6.4 BINSPECT

The BINSPECT system [Esh12] by Eshete et al is a novel lightweight web page analysis system. Like the aforementioned Prophiler, BINSPECT analyses a web page through the examination of a variety of features including: URL, HTML+JavaScript source, and social reputation. The JavaScript analysis combines static and dynamic approaches using statistical learning methodology. The results of all the features analyses are combined

utilizing a genetic algorithm to determine if the page is malicious or benign. Similar to Wepawet, BINSPECT executes the JavaScript in some form to aid in its analysis.

2.7 Conclusion

This Chapter discussed the fundamental concepts on which the researchers designed the experimentation of clone detection on JavaScript malware. It also discussed related works and explains why they were not sufficient to meet the research aim. The next chapter will discuss how these technologies were used to build a corpus of JavaScript to test a modified NiCad.

Chapter 3

Approach and Experimental Setup

3.1 Introduction

This chapter is a discussion of the design and implementation of the experiments conducted using the NiCad clone detector. The first section discusses the general experimental approach and setup of NiCad, including the initial explorations in JavaScript malware. These led to the final two full experiments introduced in the second section and discussed in full in Chapter 4 and 5, respectively.

3.2 General Approach

The general approach of our malware detection system is shown in Figure 3.1. PDF documents, containing benign or malicious JavaScript, are obtained from the VirusTotal database. The JavaScript embedded in the files is extracted into source files. The source files are then passed to the clone detection phase that varies between experiments. Two experiments are performed, the clone signature experiment and the reflexive call experiment. The NiCad Clone Detector is used in the clone detection phase in several roles. It is used to identify clone classes for the signature detection experiment, and it will be used in cross-clone mode in both experiments.

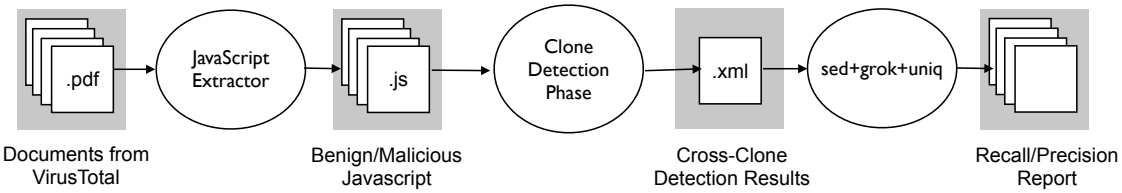


Figure 3.1 - General Process Diagram

This section discusses initial exploration and evaluation. The first sub-section discusses collecting PDF documents for analysis and the subsequent JavaScript extraction. The next sub-section gives a general description of the modifications to NiCad as it was used during experimentation. The section concludes with a discussion of the initial CVE number-based experiment.

3.2.1 Obtaining PDF Documents

For the process of modifying and evaluating NiCad for malicious JavaScript detection, 100 malicious and 50 benign PDF documents were downloaded from VirusTotal. All malicious files were downloaded using the following search parameters: "`type:pdf positives:5+ tag:js-embedded`". This selects all PDF files submitted to the archive in the last 30 days with at least five antimalware detections and a confirmation that the file contains embedded JavaScript (by VirusTotal's automatic analysis). All benign files were downloaded using the search parameters: "`type:pdf positives:0 tag:js-embedded`". The JavaScript extracted from the benign PDF sets was then manually examined to ensure that no malicious behavior was present in the scripts. The initial batch of malicious PDFs were also manually inspected and classified by their stage

1 unpacker. These hand-evaluated malicious files would be used to adjust NiCad to target JavaScript malware.

3.2.2 Modification of NiCad

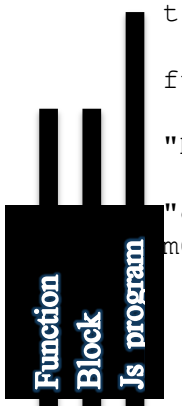
NiCad version 3.5, the latest version available since March 2013, was used during the experiments [Nic13]. NiCad was extended using its plug-in system to support JavaScript source code. Minor adjustments to configuration and NiCad's TXL launch scripts were made to tailor it for detection of malicious code [Cor06].

In order to add support for JavaScript to NiCad, we used a JavaScript grammar from previous research [Gam12] and we wrote TXL transformations for extraction, renaming, filtering, and abstraction. These transformations were created for both the default block and function syntactic elements. Blocks in c-style languages such as Java and JavaScript are defined as the statements between '{' and '}' characters.

Initial tests our JavaScript NiCad plugin on a sample of malware led to the conclusion that the default block and function syntactic elements were inappropriate for JavaScript malware. Unlike most procedural languages, it is common practice in a script to have a significant portion of the logic at the global level, not encapsulated by any function or block construct. Most of the malware we extracted from the hand-inspected PDFs made no use of function definitions; statements were entirety located at the global level. When the block and function extractors were tested, very little of the malware was extracted.

To rectify this issue, a new extractor was written that targeted the code at the global level. The syntactic element `js_program` was used instead of the block and function elements to extract potential clones.

Figure 3.2 shows a comparison of the extracted fragments. The bars to the left of the code illustrate the lines that will be retrieved by each extractor. In particular, the highlighted line that contains the payload is only extracted by the js_program extractor. For the purposes of the experiment all scripts were extracted using the `js_program` extractor.




```
t = event;

function eis(s) {
    src_table =
"MFRH3A2G1y_?BQL.&Y5aj8#PeC6%boTwxc=kJ4-l:g/";
    dest_table =
"dq&EpgKF3twh_NvQJkzB%MCo8saD0Wy:#u-
mGSO2b7A4fULVR/iX6ej1c=nP.?9HxI5lrZYT";
    ret = [];
    for (i = 0; i < s.length; i++) {
        var index = src_table.indexOf(s[i]);
        if (index > -1) {
            ret.push(dest_table[index]);
        }
    }
    return ret.join('');
}

function fff(z) {
    return eis('3VlieJoy')
}
cxz = eis('evV-');
t = t[eis('yVlAey')];
if (t.filesize > 3100) a = 'payloadgoeshere';
if (t.filesize > 3100) dsa = k = t[cxz];
s = "";
z = a;
str = eis('dyl8oA');
ss = k(str);
ff = eis('.lt0EDVlEtMe');
xz = a.length;
q = k(fff());
ss = ss[ff];
for (i = 0; xz > i; i = 2 + i) {
    var i2 = i + 1;
    var z2 = z;
    if (z[i] != '@')
        s = s+(ss(q(z[i].concat(z2[i2]),24+2)-15));
}
z = xvasvs = s;
k(z);
vdgweh = 041;
```

Figure 3.2 – Visual representation of the coverage of three extraction techniques

Supplementing the required additions to NiCad, custom configuration files were created to run the experiments. During experimentation, some normalization transforms would fail, as the default memory allocation to the TXL program was insufficient for the large intermediate XML files created from potential clone extraction. NiCad's shell scripts that call the TXL program were modified to pass on an increased memory allocation to support the large files being read into the system.

NiCad is a near-miss clone detector that has a variable difference threshold for evaluating possible clone pairs. The default similarity threshold is 70%. Most malware groupings had a similarity between 80% and 100%. After experimentation with a set of manually classified families of malware, the best results were found with a similarity of 70%, catching minor structural differences and obfuscation in malware. This similarity threshold was used in every instance of NiCad in all experimentation.

When testing in cross clone mode, NiCad is run to compare a 'set' and a 'lib' folder. The set folder contains the evaluation set of JavaScript and the lib folder represents the training set of known malware. After execution, the cross clones' XML-based report was read by the testing suite. This XML report was parsed by the sed text transformation tool to extract all 'set' folder filenames [Fre13]. The list of filenames is compared to a known list of all files in the 'set' folder using Grok to determine how many true/false positives were detected by the clone detection [Gro13].

3.2.3 CVE-based Exploration

We decided to gather files associated with specific Common Vulnerabilities and Exposures (CVE) numbers to determine if clone detection could be successfully used to find all of the members of a given family of malware. Therefore, in addition to the standard malicious and benign queries made to VirusTotal, we conducted two targeted searches to download all files tagged with a specific CVE number: CVE-2007-5659 and CVE-2013-0640 were selected. Both exploits took advantage of Adobe Acrobat Reader's JavaScript engine vulnerabilities. In total, 133 PDFs were downloaded. (100 tagged with CVE-2007-5659 and 33 tagged with CVE-2013-0640).. After preliminary manual inspection and basic clone class detection, it was discovered that the PDF files marked by the same CVE number did not maintain any consistency in their immediately accessible first stage of JavaScript (i.e. the stage 1 unpackers). The malware developers use various combinations of unpackers to hide the specific exploit.

Thus the files were categorized by the exploit, and not by the visible unpacker. Without dynamic inspection of all stages of the JavaScript malware, it would be very difficult to categorize a collection of malware into its constituent exploits. This meant that the proposed CVE-based classification system was not sufficient, and this research replaced it with a new experiment to detect families of stage 1 unpackers.

3.3 Experiment Overview

This section provides an overview of the two experiments performed using NiCad. The first uses NiCad as a clone detector to find clones of a trained sub-set of malware in a

larger evaluation set. The second experiment, uses NiCad as a pattern matching engine to find malicious reflexive calls in the same evaluation set.

3.3.1 Clone Signature Experiment

The clone signature experiment replaced the original CVE-classification scheme. In this experiment we will instead use the clone detector to classify the malware by the visible state 1 unpacker. Figure 3.3 shows the process of this experiment. We downloaded two sets of PDFs from VirusTotal: a set of malicious PDFs and a set of benign PDFs. We then extracted the JavaScript from these PDF documents and separate the malicious JavaScript into a larger *malicious evaluation set* and a *malicious training set*.

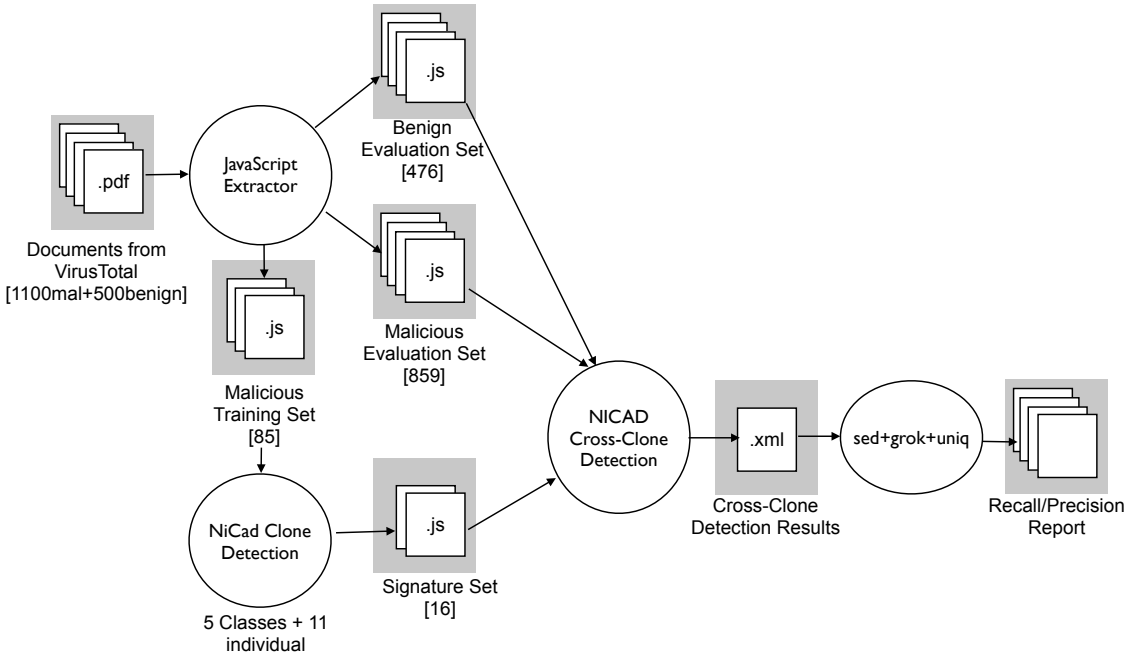


Figure 3.3 - Clone Signature Experiment Process

NiCad was used to find the clone classes within the malicious training set. One member of each clone class is then selected to form a *signature set*. We then use NiCad in cross-clone detection to find clones of the members of the signature set in the malicious and benign evaluation sets evaluating various parameters to the detector

Figure 3.4 provides a graphical overview of the data sets used for the Clone Signature experiment with numerical values corresponding to the number of successfully extracted JavaScript files.

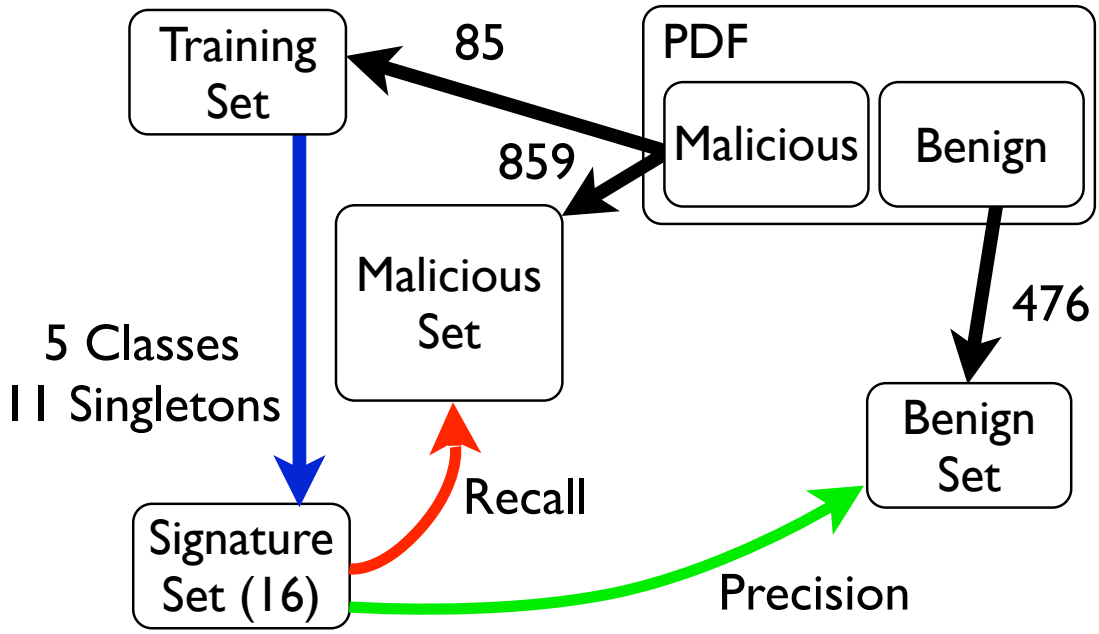


Figure 3.4 – Clone Signature Experiment JavaScript Sets

The malicious set consisted of 1100 PDF files containing JavaScript using the query tags `type:pdf positives:5+ tag:js-embedded`. 100 files from this set are randomly selected and assigned as the malicious training set. The remainder of the files is used as

the *malicious evaluation set*. The second set was a set of 500 benign PDFs containing JavaScript selected using the query `type:pdf positives:0 tag:js-embedded`. This *benign evaluation set* is used as a control set.

Both Pita-J, and libPDFjs were used to extract JavaScript from the malicious and benign evaluation sets [Las11]. Results are shown in Table 1. Of the 1000 PDF files in the malicious evaluation set, only 451 had their JavaScript extracted successfully by libPDFjs whereas Pita-J extracted 859 files. For the 500 PDF documents in the benign evaluation set, 476 files were successfully extracted by libPDFjs and only 245 files were extracted by Pita-J. For our experiment, we used the malware JavaScript extracted by Pita-J and the benign JavaScript extracted by libPDFjs. The malicious training set of 100 PDFs resulted in 85 successfully extracted files.

Table 1 - Comparison of PDF extractor performance

	Malicious Evaluation Set		*Benign Evaluation Set*		*Malicious Training Set*	
	Files	**Percentage**	**Files**	**Percentage**	**Files**	**Percentage**
Pita-J	859	***86%***	245	***49%***	85	***85%***
libPDFjs	451	***45%***	476	***95%***		
Total files	1000		500		100	

We run cross-clone detection between the signature set and both the malicious and benign evaluation sets to calculate recall and precision respectively.

3.3.2 Reflexive Call Experiment

As a result of the findings in the clone signature experiment, the presence of `eval` statements in malware was further examined. Most malicious scripts would use reflexive

calls to occlude their use of the essential function. While reflexive calls are not uncommon in more complex web applications, there is less reason to use them in the smaller programs contained in PDF documents. We therefore hypothesized that the presence of reflexive calls in PDF documents is a good indication of their maliciousness. Reflexive calls have a unique structure that can be detected using pattern matching after normalization. The second experiment modifies NiCad to act as a pattern matcher for reflexive calls in JavaScript source. Figure 3.5 presents the process of the reflexive call experiment.

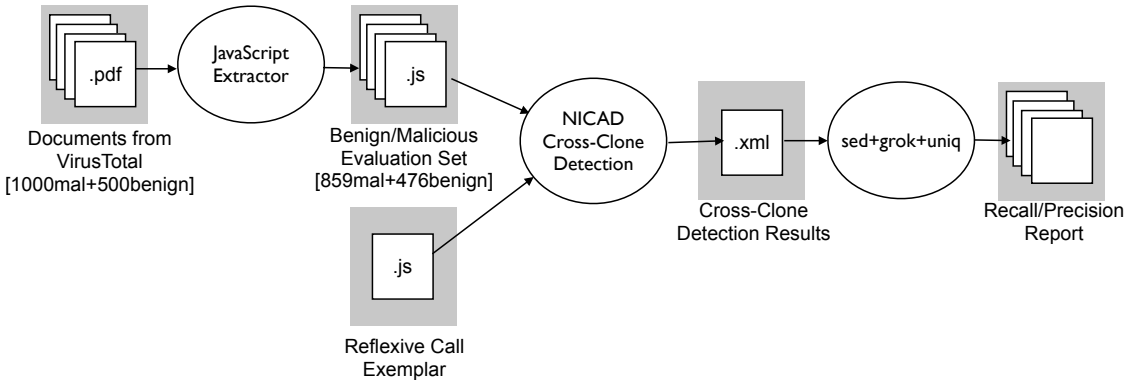


Figure 3.5 – Reflexive Call Experiment Process

A specifically tailored normalization and extraction transformation was written to reduce processing time of the NiCad system. The same malicious and benign evaluation sets from the clone signature experiment are used to compare detection performance between the two detection approaches. These evaluation sets are cross-clone detected against a reflexive call exemplar. The reflexive call experiment did not detect as many malicious files as the clone signature experiment but it detected files that were entirely by the clone

signature experiment. Also, it should be noted that this alternative detection mechanism required no signature library and no upkeep. We suspect that such detection techniques would deter future malware from using these extractors.

3.4 Conclusion

This chapter discussed the design of the experiments using the NiCad clone detector. A brief overview of the discarded CVE-based experiment was also given, including a description of the need for our custom-built Pita-J extractor. The chapter also introduced the two experiments carried out by this research. The following two chapters detail the analysis of results for those two major experiments.

Chapter 4

Analysis and Results - Clone Signature Detection

4.1 Introduction

This chapter discusses the first complete experiment on the NiCad clone detector as a malware detection tool on malicious JavaScript in PDF files. As explained in the previous chapter, the clone signature detection experiment uses a training malicious set of PDF-borne JavaScript to cross-compare with the larger malicious evaluation and benign evaluation sets. The first two subsections discuss the modifications to NiCad's extraction and normalization systems required by the experiment, respectively. Finally, results of the experiment are discussed in Subsection 4.3.

4.2 Experiment Methodology

The experiment is meant to determine if stage 1 unpackers can be detected by using NiCad to find the clones in the malicious training set. The results of the clone detection are used as a seed to cross detect clones in the malicious evaluation set. The `js_program` syntactic boundary is used for extracting potential clones and normalization rules are varied to find the best combination of filtering and abstraction. To gauge the accuracy of the detection, the same malicious training set is applied against the benign evaluation set.

We first run the NiCad on the malicious training set to identify similar malware programs. The results are displayed in Table 2. We used blind renaming and abstraction of literals as normalization rules to maintain the structure of the malware. The detection found five clone classes containing 74 of the files from the malicious training set. Most sets/clone classes had 100% similarity after renaming and abstraction, however one set/clone class had a similarity of only 83%. After examination, the difference was found to be in the payload and not in the unpacker. This exercise demonstrated the extent of reuse of stage 1 unpackers in malware currently distributed online. Since all of the members of each clone class are similar, comparing all members of the clone class to the malicious evaluation set is redundant. Thus a smaller set, the *signature set*, was compiled from one file from each of the five clone classes and the 11 singleton files that were not categorized for a total of 16 JavaScript files.

Table 2 – Training Set Results

Clone Class	*Files*	*Similarity*
1	2	83%
2	2	100%
3	2	100%
4	63	100%
5	5	100%

4.2.1 Normalization Rules

Various normalization rules were tested to compare malicious and benign detection performance. Because some JavaScript malware consists of a single monolithic statement

(formatted as one line by the JavaScript grammar), the minimum clone size was changed from the default 10 lines to 1 line.

Four different normalization rules were used along with no normalization for the evaluation:

Blind Renaming

This setting replaced all identifiers including variable and function names with an '`x`' character. All other normalization rules include blind renaming as a first stage of normalization.

Blind Renaming and Abstraction of Literal Values

This setting replaces all literal values with the identifier `js_literal` along with blind renaming of names. This abstraction of `js_literal` removes deviations in stage 2 payloads and other variables that could be stored in-line in the stage 1 unpacker. Figure 4.1 demonstrates a stage 1 unpacker that contained an in-line payload that was assigned to a variable being abstracted to `js_literal`.

Before Abstraction

```
for (x = 128; x > = 0; - - x) x + = x;
var x =
'\u46EB\u315F\u83C9\u01E9\uFE89\uC030\u012C\uAEF2\u47FE\u
89FF\u30FB\u2CC0\uF201\uFEAE\uFF47\uFD89\uAEF2\u47FE\uEBF
F\u6071\uC931\u8B64\u3071\u768B\u8B0C\u1C76\u5E8B\u8B08\u
2056\u368B\u3966\u184A\uF275\u5C89\u1C24\uC361\u5BEB\u8B6
0\u246C\u8B24\u3C45\u548B\u7805\uEA01\u4A8B\u8B18\u205A\u
EB01\u34E3\u8B49\u8B34\uEE01\uFF31\uC031\uACFC\uC084\u077
4\uCFC1\u0112\uEBC7\u3BF4\u247C\u7528\u8BE1\u245A\uEB01\u
8B66\u4B0C\u5A8B\u011C\u8BEB\u8B04\uE801\u4489\u1C24\uC36
1\u54EB\uD231\u5252\u5553\uFF52\uEBD0\uEB1A\uE85D\uFF7B\u
FFFF\uE7BA\u8BBA\u52C4\uE850\uFF92\uFFFF\uD231\uFF52\uEBD
0\uE82D\uFF63\uFFFF\uAABA\u8A6E\u52F3\uE850\uFF7A\uFFFF\u
D231\uC283\u83FF\uFAEA\u5352\uD0FF\uC9EB\u47BA\uC87D\u52A
0\uE850\uFF60\uFFFF\uAEEB\u5DEB\u34E8\uFFFF\uBAFF\uCE12\u
091A\u5052\u4BE8\uFFFF\u56FF\uD0FF\uDAEB\uF9E8\uFFFE\u75F
F\u6C72\u6F6D\u2E6E\u6C64\uFF6C\u2E2E\u752F\u6470\u7461\u
2E65\u7865\uFF65\u7468\u7074\u2F3A\u772F\u7777\u772E\u6E6
1\u732E\u7274\u7461\u6765\u6F79\u6A62\u6365\u2E74\u6F63\u
2F6D\u6F64\u6E77\u6F6C\u6461\u665F\u6C69\u2E65\u6870\u3F7
0\u3D65\u6441\u626F\u2D65\u3032\u3830\u322D\u3939\uFF32\u
03CD';
```

After Abstraction

```
for (x = js_literal; x > = js_literal; - - x) x + = x;
var x = js_literal;
```

Figure 4.1 - Effect of abstracting in-line payloads

Blind Renaming and Filtering of Assignment Expressions, and Variable Declarations

This setting removes all assignment and declaration statements leaving only control statements (such as `if` and loop statements). However, due to the structure of the TXL grammar for JavaScript, removing these statements did not remove the trailing semicolon

delineating the end of the statement. This meant that the normalization retained the original length of the potential clone but not its contents. Figure 4.2 gives an example of JavaScript before and after this normalization.

Blind Renaming and Filtering of Assignment Expressions, Variable Declarations, and Arguments (Control only)

This setting builds on the previous filtering normalization by also removing the semicolons. This leaves exclusively the control statements. Figure 4.2 demonstrates the difference between this filtering setting and the previous, semi-colon preserving setting.

Before Filtering

```
xx = 'b';
a = 'PAYLOAD';
s = '';
l = 'al';
if ((event.name + '').substr (0, 3) = = 'O' + 'pe') {
    t = event;
}

p = t.target [String.fromCharCode.apply (String, [112,
97, 114, 115, 101, 73, 110, 116])];
for (i = 0; i < a.length; i + = 2) {
    jj = 0;
    s + = String.fromCharCode (p (a.substr (i, 2), 30 +
1));
}
t.target [(xx = = 'b') ? ('ev' + l) : 0] (s);
```

After Filtering of Assignment Expressions, and Variable Declarations

```
;
;
;
;
if () {
    ;
}

;
for (;;) {
    ;
    ;
}
;
```

After Filtering of Assignment Expressions, Variable Declarations and Arguments (Control Only)

```
if () {
}
for () {
}
```

Figure 4.2 - Comparison of Filtering Settings

4.3 Results

The experiment is mainly evaluated through two parameters, recall and precision, based on the two evaluation sets (malicious evaluation set and benign evaluation set).

Recall is the fraction of all relevant files retrieved by a query. It is a measure of how many documents are missed. It is defined as [Rij79]:

$$recall = \frac{|\{relevant\ files\} \cap \{retrieved\ files\}|}{|\{relevant\ files\}|}$$

In our context, the relevant files are the files in the malicious evaluation set. The retrieved files are those that are those that NiCad identifies as clones of the signature set. Thus recall measures the fraction of the malicious evaluation set that was identified by NiCAD.

Precision is the fraction of relevant documents retrieved by a query. It measures how many irrelevant documents are retrieved in error. It is defined as [Rij79]:

$$precision = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{retreived\ documents\}|}$$

In our context, precision is less than 100% when any of the benign files are identified as clones of the signature set.

The F-measure , or accuracy, of a query is the harmonic mean of its precision and recall. It is a weighted average between the precision and the recall. It is defined as [Rij79]:

$$F = 2 * \frac{recall * precision}{recall + precision}$$

The results of cross clone detection between the signature set and the evaluation sets are shown in Table 3. These results are calculated against all 1500 malicious and benign documents. Those files that could not be extracted by libPDFjs and Pita-J were considered non-suspicious by our approach.

A significant portion of the files in the malicious training set (39%) are detected without any normalization. Blind renaming alone did not improve detection. These 393 files were not always identical clones of files in the signature set, but shared at least 70% of their lines of code. Adding further normalization techniques to blind renaming leads to a 75% detection rate. Despite the differences in approach, both filtering and abstraction-based normalization resulted in near identical detection performance. Filtering out all but control statements was slightly more effective by detecting three more files. In terms of accuracy, the overall best normalization was found to be blind renaming and abstraction at accuracy of 86%.

Table 3 – Comprehensive Signature Detection Results

Parameters	*Malicious Files w/ Clones*	*Benign Files w/ Clones*	*Recall*	*Precision*	*Accuracy (F-Measure)*
no normalization	393	0	39%	100%	56%
blind renaming	393	0	39%	100%	56%
blind renaming + abstract literals	747	0	75%	100%	86%
blind renaming + filter assign and decls	747	56	75%	89%	81%
blind renaming + filter assign, decls, args	750	5	75%	99%	85%
Total Files in Set	**1000**	**500**			

If we ignore the 165 files that were not extracted by libPDFjs and Pita-J tools and consider only the effectiveness of the clone detection phase of our process, we get somewhat better results, shown in in Table 4. These results show the performance of the our system without the extractor's influence, concentrating on the performance of the clone detection. Precision values remain unchanged, as the removed files were already marked as benign previously. Recall and accuracy results rise and blind renaming and abstraction reaches 87% recall and 93% accuracy.

Table 4 – Reduced Signature Detection Results

Parameters	*Malicious Files w/ Clones*	*Benign Files w/ Clones*	*Recall*	*Precision*	*Accuracy (F-Measure)*
no normalization	393	0	46%	100%	63%
blind renaming	393	0	46%	100%	63%
blind renaming + abstract literals	747	0	87%	100%	93%
blind renaming + filter assign and decls	747	56	87%	88%	88%
blind renaming + filter assign, decls, args	750	5	87%	99%	93%
Total Extracted Files in Set	**859**	**476**			

Manual examination of the malicious evaluation set showed that the files that were not detected by cross-clone detection belonged to malware families unrelated to the ones in the signature set with a few exceptions resulting from syntax errors and 1 genuinely benign file.

Benign files are only misidentified as malicious when assignments and declarations are filtered. 12 percent of benign files detected are when semicolons are not filtered whereas only 1 percent is detected when they are also filtered. The removal of the semicolon from the clone evaluation increased precision, as it no longer preserved the size of potential clones. Upon inspection of the false positives, the files incorrectly detected as malicious were of similar size to the malicious files even if their control structures were different. Those control statement differences did not reduce the similarity index below the 70%

threshold and were therefore marked as clones by NiCad. Filtering all but the control statements still led to incorrect classification of 5 benign files as malicious.

The most efficient normalization technique was found to be blind renaming plus abstraction of literals. This methodology detected 87% of total malicious files while still maintaining 100% precision.

To determine why 139 successfully extracted files were not detected by NiCad at any normalization, the false negative reporting files were manually inspected. 93 files were very similar in structure to one clone class of detected malware. However, these files could not be properly parsed by the basic TXL extractor. Upon inspection, it was discovered that these files had syntax errors. These errors could have been caused by either a corrupted PDF extraction or inherent glitches in the PDF prior to delivery. These syntax errors, an example of which is shown in Figure 4.3, should cause the JavaScript interpreter in the PDF reader to fail. Some other files which were reported as false negatives had TXL parsing errors due the JavaScript grammar's definition of in-line regular expressions. One file was found to contain no malicious code, which is most likely a sign of a malicious PDF that uses a vector other than JavaScript for exploitation rather than an incorrect classification by VirusTotal. Most other files that were reported as false negatives were legitimate files that were dissimilar to any items in the signature set.

```
z=c['s'+"ubstr"](0,1);
s[a](z);
z=c['s; //Syntax error
s[a](z);
z=c['s'+"ubstr"](69,1);
s[a](z);
z=c['s'+"ubstr"](73,1);
```

Figure 4.3 - Snippet of syntax error in misclassified code

4.4 Conclusion

This chapter discussed the configuration and results of the clone signature experiment. The experiment was found to be fairly effective at detecting malware with 87% recall and 100% precision when the abstraction normalization was used. The clone signature experiment models a traditional anti-malware use case where updates are be made to the signature set of malware to detect new malware types. To gauge if the leftover 13% of malware could be detected through a method that did not require such manual intervention, a second experiment was attempted that searches for a specific pattern. This experiment is discussed in Chapter 5.

Chapter 5

Analysis and Results – Reflexive Call Detection

5.1 Introduction

A second experiment was conducted to determine if NiCad could be used effectively as a pattern matcher for reflexive calls, which are, as we previously discussed, a very obvious indicator of malicious intent.

5.2 Reflexive Calls in Malware

All unpackers must, invariably, run the `eval` function to execute their payload. As the `eval` function has no legitimate use in JavaScript for PDF documents, malware detection tools search for the keyword `eval` as an easy indication of malicious intent. Malware developers have adjusted to this by avoiding explicit calls to the `eval` function. Reflexive calls in JavaScript allow malware developers to build the keyword `eval` in piecemeal, obfuscated form and execute it by transitively loading the reassembled string at runtime. An example of one such malware sample is provided in Figure 5.1. The operation of this code was discussed in Section 2.4.2.

```
var pr = null;
var fnc = 'ev';
var sum = '';

app.doc.syncAnnotScan();

if (app.plugIns.length != 0) {
       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(/-/);
       var proc = String.fromCharCode(22+15);
       for (var i = 1; i < arr.length; i++) {
              buf += String.fromCharCode("0x"+arr[i]);
       }
}

if (app.plugIns.length >= 2) {
       fnc += 'l';
       app[fnc](buf);
}
```

Figure 5.1 - Sample Stage 1 Unpacker – Also seen in Figure 2.15

5.2.1 Reflexive calls

The syntactic structure of a reflexive call in JavaScript has three parts. The first portion is an object that hosts the function call itself. The `app` keyword or a self-referencing `this` is often used, sometimes after being assigned to a variable. The second portion enclosed by the `[` and `]` tokens contains a expression that evaluates to the string value `'eval'`.

The third portion of the reflexive call is the parameter for the `eval` call itself. In most cases, this is the second stage payload that has been retrieved and decoded prior to the reflexive call. In the Figure 5.1, the `fnc` variable stores the `'eval'` string and the `buf` variable stores the decoded JavaScript function that contains the exploit payload. This unique three-stage structure of a reflexive call allows for quick pattern matching after normalization.

5.3 Modifications to NiCad

NiCad was modified to detect reflexive calls by adding an extractor that targets the syntactic element for primary expressions (`js_primary_expn`). The use of blind renaming and some additional filters allows us to use a singular file to define the desired reflexive function calls. In this form, NiCad is not acting as a clone detector but rather as a pattern-matching engine. Its renaming and filtering mechanisms are used to enhance pattern detection.

5.3.1 Extraction

The syntax for reflexive call in the JavaScript TXL grammar used in NiCad is shown in Figure 5.2. A reflexive call is a special type of `js_primary_expn` with at least two `js_selector`s appended. A new syntactic unit called `js_extended_primary` was defined to identify such a syntactic element. `The js_member_expn` element of the grammar was redefined to parse all instances of `js_primary_expn` with two or more `js_selector`s as a `js_extended_primary`. It should be noted that `js_selector`

parses all selectors including `js_subscripts` (for the '`[ ]`' syntax) and `js_arguments` (for the '`( )`' syntax). This means that many other expressions, along with reflexive calls, will be extracted by using this syntactic boundary.

```
Single-line Reflexive Call
app[eval_string](payload_as_parameter;
SyntacticSyntactic Element View in TXL
[js_primary_expn][js_subscripts][js_arguments]
```

Figure 5.2 – Reflexive call syntax in TXL

5.3.2 Normalization

After extraction, the reflexive call must be normalized to match the expected reflexive call that is being cross-compared (discussed in the next section). The filtering normalization used in the clone signature detection experiment removed assignment and declaration expressions, which was expected suffice in removing all but the structure of the call. However, upon testing the filtering, some vestigial elements remained in the potential fragments that led to misclassification.

First, the leading primary expression could have multiple field selectors. The field selectors (delimited by the '`.`' Character) were filtered as shown in Figure 5.3.(c). Next, filtering expressions and declarations removes most discrepancies as shown in Figure 5.3. (d). Additionally, the parameters of the reflexive call could have multiple arguments separated by the '`,`' character. Filtering the commas required a redefinition of

`js_arguments` to add a new nonterminal `js_arguments_list` to separate the list of arguments from the `(` and `)` characters. The result is an expression pruned to the minimal characters needed to recognize the structure of the expression as shown in Figure 5.3.(e)

```
No normalization (a)
a.b.c[d + "l"](payload_as_parameter, options)
Blind Renaming (b)
x.x.x[x + "l"](x,x)
Filtering of field selector (c)
x[x + "l"](x,x)
Filtering of expression and declarations (d)
x[](,)
Filtering of arguments list (e)
x[]()
```

Figure 5.3 - Demonstration of Reflexive call filtering

5.3.3 Malicious Patterns

Figure 5.4 shows the entire set of malicious patterns to match for the reflexive call experiment. Originally, the file consisted of a single line that had the desired reflexive call structure. However, the keyword `this` can not be renamed and therefore, a second line was added to the comparison pattern to account for reflexive calls that use the 'this' keyword.

```
x [eval] ("somecodegoeshere");
this.x [x] (x);
```

Figure 5.4 - Malicious pattern set

5.4 Results

Table 5 shows the results of the pattern matching of reflexive calls in the same malicious evaluation set used in the clone signature experiment (Chapter 4). This is a comprehensive evaluation of the system with the extractor process factoring into the recall and precision metrics. As expected, no files are detected when no normalization is applied or only blind renaming is used. When the filtering rules, discussed earlier in Subsection 5.3.2, are enabled a total of 566 files (57%) of total malicious files are classified as malicious by NiCad. In comparison, the clone signature experiment achieved 75% recall using abstraction of literals. In this case, no benign files were found to contain the reflexive call that was being targeted. This was an expected result as the usage of such calls are very unlikely in PDF documents that do not try to obfuscate malicious intent.

Table 5 – Comprehensive Reflexive Call Detection Results

Parameters	*Malicious Files w/ Pattern*	*Benign Files w/ Pattern*	*Recall*	*Precision*	*Accuracy (F-Measure)*
no normalization	*0*	*0*	0%	100%	0%
blind renaming	0	0	0%	100%	0%
blind renaming + filter assign, decls, args, selectors	566	0	57%	100%	72%
Total Files in Set	**1000**	**500**			

Table 6 only considers the total successfully extracted files and has adjusted recall and precision values. These values remove the focus the performance values on the pattern matcher by removing the influence from the extraction tools. The pattern matcher's recall increases to 66% and accuracy rises to 80% from 72%.

Table 6 – Reduced Reflexive Call Detection Results

Parameters	*Malicious Files w/ Pattern*	*Benign Files w/ Pattern*	*Recall*	*Precision*	*Accuracy (F-Measure)*
no normalization	0	0	0%	100%	0%
blind renaming	0	0	0%	100%	0%
blind renaming + filter assign, decls, args, selectors	566	0	66%	100%	80%
Total Extracted Files in Set	**859**	**476**			

The reflexive call experiment missed 34% of all files in the malicious evaluation set, even if most use an `eval` function call during operation. There are three possibilities for these missed detections. First, and very rarely, some malicious PDFs do not contain any unpackers. The exploit is found in the first stage of JavaScript that is launched by the PDF reader. This is very unlikely and was not exhibited in any of the files present in the malicious evaluation set. These files can be easily detected through signature detection without the aid of more sophisticated approaches.

The second option is that there was no obfuscation to hide the '`eval`' keyword. This case was found in a few files in the malicious evaluation set, but is seldom the case.

These files can be detected by plaintext searches for the 'eval' keyword, requiring significantly less execution time than the techniques discussed in this paper.

The third, and most likely possibility is that the reflexive call itself has been further obfuscated. Malware developers can split the three part single-line reflexive call into multiple lines. For example, the app[eval_string] function object of the reflexive call can be first assigned to a new variable and then be invoked on a separate line with the parameters to be passed to eval. Figure 5.5 demonstrates the differences between a single-line reflexive that the experiment can detect and a multi-line reflexive call. This multi-line reflexive call cannot be detected by syntactic pattern matching. Data flow analysis is needed to distinguish the reflexive retrieval of a function object from other values.

```
Single-line Reflexive Call
app[eval_string](payload_as_parameter;
Multi-line Reflexive Call
var temp = app[eval_string];
temp(payload_as_parameter);
```

Figure 5.5 - Comparison of Single and Multi-line Reflexive Calls

5.4.1 Combined Recall

While the reflexive call experiment did not match the performance of the clone signature detection experiment, it detected malicious files that the other approach missed. Some of the files in the malicious evaluation set did not have a similar file in the malicious

training set. This is similar to a zero-day vulnerability where a new class of malware exists where no signatures are available to detect it. Combining the techniques in a sequential malware detection system would first test clone signatures for known classification. Should no known classification exist, the reflexive call system could look for a telltale signs of malicious intent. This type of sequential system would have detected 99% of the total malicious evaluation set with no false positives after removal of extractable files (Table 8).

Table 7 – Comprehensive Combined Results

	Clone Signature *blind renaming + abstract literals*	***Reflexive Call*** *blind renaming + filter assign, decls, args, selectors*
Malicious Files w/ Clones	745	566
Benign Files w/ Clones	0	0
Recall	75%	57%
Precision	100%	100%
Accuracy	86%	72%
Unique Malicious Files	284	104
Unique Benign Files	0	0
Combined Malicious Detected		**850**
Total Malicious Files		**1000**
Combined Benign Detected		**0**
Total Malicious Files		**500**
Combined Recall		**85%**
Combined Precision		**100%**
Combined Accuracy		**92%**

Table 8 – Reduced Combined Results

	Clone Signature *blind renaming + abstract literals*	***Reflexive Call*** *blind renaming + filter assign, decls, args, selectors*
Malicious Files w/ Clones	745	566
Benign Files w/ Clones	0	0
Recall	87%	66%
Precision	100%	100%
Accuracy	93%	80%
Unique Malicious Files	284	104
Unique Benign Files	0	0
Combined Malicious Detected		**850**
Extracted Malicious Files		**859**
Combined Benign Detected		**0**
Extracted Benign Files		**476**
Combined Recall		**99%**
Combined Precision		**100%**
Combined Accuracy		**99%**

5.5 Conclusion

This chapter discusses an experiment that repurposes NiCad to act as a pattern matcher to detect reflexive calls found only in malicious JavaScript. The results demonstrated a recall of 66% with 100% precision. While this performance is not as good as those of the clone signature experiment (chapter 4), the reflexive call experiment found previously

undetected malicious files. A proposed combined system that uses the reflexive call system as a fallback when the clone signature detection fails, had 99% recall and 100% precision.

Chapter 6

Conclusion

6.1 Contributions

The research aim of determining the feasibility of clone detection techniques in detecting script-based malware was achieved by the clone signature and reflexive call experiments on NiCad and JavaScript. The result of the combined experiments produced a 99% detection rate with 0 false positives prove that clone detection techniques can produce excellent results in malware detection.

Four different syntactic boundaries were examined to determine which would be most effective for detecting PDF JavaScript malware. As discussed in Section 4, the `js_program` syntactic element was determined to represent the best abstraction for the small JavaScript files under investigation. Parameters were modified in NiCad to determine their effect on the accuracy and recall of classifying PDF malware with a known set of exemplars. In Section 4.3, the blind renaming and abstraction of literals normalization set was found to combine a high (75%, or 87% after removing PDF extraction errors) detection rate with no false positives on a set of malware when trained with a subset. The clone detection tool NiCad's readjustment as a pattern-matching engine for malware generated lower results with 57% (66% after removing PDF extraction errors) of total malware detected. However, it was hypothesized that a proposed combined system utilizing both techniques could achieve 85% (99% after

removing PDF extraction errors) detection rate with no false positives. The research also highlighted deficiencies in the categorization of malware. The initial CVE-identifier experiment failed because of the single dimensional CVE number's inability to reflect the class of unpacker within each file. Malware organization, along with CVE classification, should take into account the stage 1 unpacker or obfuscation technique employed by each file. Adding the family of the stage 1 unpacker as a second dimension to malware classification could aid security researchers' comprehension of the state of current malware.

6.2 Future Work

This research endeavor has proven to be valuable, as evidenced by the abundance of the future work described below. The possibilities for further research include increases in scope and depth within the subject of malware detection based on clone detection.

6.2.1 Increase the Breadth

The experiments conducted had a working set of 1100 malicious and 500 benign for a total of 1600 PDF files. Increasing the number of files could help in discovering many more obfuscation techniques employed by JavaScript malware developers. In addition, the Clone Signature experiment can be refined by repeating the experiment with a different randomly selected malicious training set.

6.2.2 More normalization

NiCad offers more normalization options beyond the filtering, renaming, and abstraction used in the experiments. Of note are the flexible pretty printing capabilities of NiCad. This normalization technique allows for fine-grained detection by splitting single line constructs into multiple lines. Each line then has equal effect on the similarity metric calculated during the LCS-based comparison. This sort of normalization technique could be used to accentuate the effect of certain source features that are prevalent in malware.

6.2.3 Performance evaluation

No benchmarking of performance was done during the experiment. The NiCad system was run on a variety of systems and virtual machines with no consistent allocation of computing resources. In addition, the TXL transformations to add support to JavaScript to NiCad were not precompiled, which likely had an effect on performance although this effect was not quantified. This was done because of the volatility of the development process, along with memory issues encountered when analyzing large numbers of source files. Interpreting the TXL transformations directly allowed the modification of the input memory allocation during extraction and normalization stages. Automating the execution of the PDF extractor, NiCad, and the report interpreter would allow file-by-file performance to be tested. Measuring such an automated system's resource utilization and execution time would help gauge the feasibility of this system in various real-world tool scenarios.

6.2.4 Implementation

In the two experiments NiCad was used as a bulk comparison system that compared the malicious and benign evaluation sets en masse instead of a more realistic check of a single file as would be expected in malware detection systems. Automating the entire process would allow for evaluation of this novel clone detection-based technique in conditions that more realistically approximate operational deployment. As is the case for other tools, such an automated system could be adjusted to run at the client level as an immediate protection system for users' systems. The static nature of the system's analysis mechanisms would, however, allow the execution of a NiCad-based malware detector on an intrusion detection system or other system with high security requirements.

6.2.5 Other Languages

Scripting languages other than JavaScript can also be investigated. ActionScript found in Adobe Flash files are a primary interest as Flash is a major vector of malicious attack. Any malware written in an interpreted language could be added to NiCad through the plug-in architecture.

6.3 Comparison to Related Work

Our research differs from related work in its detection methodology and implementation requirements. Our approach with NiCad and the custom PDF extractor runs outside of any reader, similar to PJScan, albeit utilizing very different detection approaches. PJScan uses a token-based machine-learning algorithm in contrast to NiCad's hybrid parser/lexical approach to classification. This also differs from Zozzle, which classifies

malware by their abstract syntax tree. Zozzle also targets web-based JavaScript in its current iteration not PDF-based JavaScript malware. Running outside of a PDF reader has the drawback of requiring the development of a PDF-parsing JavaScript extractor that roughly matches the actions of actual readers. We hypothesize that independent operation outside the reader makes our tool more valuable. Our tool can be run on firewalls and intrusion detection systems where the security policy would not allow the execution of a JavaScript engine. For PDF-based JavaScript malware, Zozzle would require integration with the PDF reader's JavaScript engine. Similar to Zozzle, Wepawet and BINSPECT also execute JavaScript code through an interpreter for analysis. Our approach only parses the JavaScript code in question; it is never executed.

6.4 Conclusion

Malware writers and security analysts will continue their cat and mouse game. Clone detection appears a promising tool for defenders. Most script-based malware is obfuscated to avoid basic lexical tools that execute keyword searches. Privacy and security are integral requirements of all digital systems and novel approaches are needed to keep data secure and the attackers at bay.

References

[Ado06] Adobe Systems Incorporated. PDF reference. Technical Report 6th Ed v1.7, 2006.

[Ado08] Adobe Systems Incorporated. The Flash Platform - Adobe. www.adobe.com/platform/whitepapers/platform_overview.pdf, Last accessed August 2013.

[Ado10] Adobe Systems Incorporated. Inside Adobe Reader Protected Mode. http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-part-1-design.html, Last accessed August 2013.

[Ber00] Bergroth, L., Hakonen, H., & Raita, T., A survey of longest common subsequence algorithms, In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on* , pp.39,48, 2000.

[Can11] Canali, D., Cova, M., Vigna, G., & Kruegel, C. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 197–206, New York, NY, USA, 2011.

[Cor06] Cordy, J. R. The TXL source transformation language. *Sci. Comput. Program.,* 61(3):190–210, 2006.

[Cor11] Cordy, J. R., & Roy, C. K. The Nicad clone detector. In *ICPC*, pages 219–220. IEEE Computer Society, 2011.

[Cur11] Curtsinger C., Livshits, B., Zorn, B., & Seifert, C. Zozzle:fast and precise in-browser javascript malware detection. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[Deu99] van Deursen A., & Kuipers T. Building documentation generators. In *ICSM*, pages 40–49, 1999.

[Ecm99] ECMA International, ECMA-262. ECMAScript Language Specification, 3rd Edition, December 1999.

[Esh12] Eshete B., & Villafiorita B. BINSPECT: Holistic Analysis and Detection of Malicious Web Pages. In *Security and Privacy in Communication Networks*, pages 149-166, 2012.

[Fre13] Free Software Foundation Incorporated. GNU sed. www.gnu.org/s/sed/, Last accessed August 2013.

[Gam12] Gama, W., Alalfi, M. H., Cordy, J. R., & Dean, T. R. Normalizing object-oriented class styles in JavaScript. In *Web Systems Evolution (WSE), 2012 14th IEEE International Symposium on*, pp. 79-83. IEEE, 2012.

[Gro13] Grok - A powerful pattern-matching/reacting tool. http://code.google.com/p/semicomplete/wiki/Grok, Last accessed August 2013.

[Hon13] Honorof, M. Chinese Hackers Take Aim at American Drones. *Tech News Daily*, http://www.technewsdaily.com/17822-hackers-aim-international-drones.html, last accessed May 2013.

[Kar13] Karademir, S.A., Dean, T.R., & Leblanc, S.P. Using Clone Detection to Find Malware in Acrobat Files. In *23rd Annual International Conference on Computer Science and Software Engineering*. CASCON 2013, (to appear).

[Lan04] Lancaster, T., & Finta, C. A Comparison of Source Code Plagiarism Detection Engines. In *Computer Science Education*, Vol. 14(2):101-112, June 2004.

[Las11] Laskov, P., & Šrndić, N. 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.

[Nic13] NiCad3 Clone Detection System, Version 3.5 (14 March 2013). http://www.txl.ca/nicaddownload.html, Last accessed August 2013.

[Nvd13] National Vulnerability Database Search: Adobe Acrobat, http://web.nvd.nist.gov/view/vuln/statistics-results?cves=on&query=Adobe+Acrobat&cwe_id=&pub_date_start_month=-1&pub_date_start_year=-1&pub_date_end_month=-1&pub_date_end_year=-1&mod_date_start_month=-1&mod_date_start_year=-1&mod_date_end_month=-1&mod_date_end_year=-1&cvss_sev_base=MEDIUM_HIGH&cvss_av=&cvss_ac=&cvss_au=&cvss_c=&cvss_i=&cvss_a=, Last accessed September 2013.

[Ora13] Oracle Corporation. Learn about Java Technology. http://www.java.com/en about/, Last accessed August 2013.

[Pit13] Pita-J. Pita-J: A JavaScript Extractor for PDF. https://cetus.ee.queensu.ca/Research/Security/Pdf/PitaJ.html, Last accessed September 2013.

[Pop13] Poppler. Poppler project website. http://poppler.freedesktop.org, Last accessed June 2013.

[Rij79] van Rijsbergen, C. Information Retrieval. Butterworths, 1979.

[Roe99] Roesch, M. "Snort: Lightweight Intrusion Detection for Networks." In LISA, vol. 99, pp. 229-238. 1999.

[Roy08] Roy, C. K., & Cordy, J. R. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *16th Int. Conf. on Program Compreh*., pages 172–181, 2008.

[Roy09] Roy, C. K., Cordy, J. R., & Koschke, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, *74*(7), pages 470-495, 2009.

[Vir13] VirusTotal. Virustotal malware repository. https://www.virustotal.com/en/about/about/, Last accessed June 2013.

[Vir13b] VirusTotal Intelligence Portal Help. Virustotal malware repository. https://www.virustotal.com/intelligence/help/, Last accessed August 2013.

[Wep13] Wepawet. http://wepawet.cs.ucsb.edu, Last accessed August 2013.