Spring 2016

Malicious JavaScript Detection using Statistical Language Model

Anumeha Shah
San Jose State University

Malicious JavaScript Detection using Statistical Language Model

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Anumeha Shah

May 2016

The Designated Project Committee Approves the Project Titled

Malicious JavaScript Detection using Statistical Language Model

by

Anumeha Shah

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2016

Dr. Thomas Austin	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science
Dr. Jon Pearce	Department of Computer Science

ABSTRACT

Malicious JavaScript Detection using Statistical Language Model

by Anumeha Shah

The Internet has an immense importance in our day to day life, but at the same time, it has become the medium of infecting computers, attacking users, and distributing malicious code. As JavaScript is the principal language of client side programming, it is frequently used in conducting such attacks. Various approaches have been made to overcome the JavaScript security issues. Some advanced approaches utilize machine learning technology in combination with de-obfuscation and emulation. Many methods of analysis incorporate static analysis and dynamic analysis. Our solution is entirely based on static analysis, which avoids unnecessary runtime overhead.

The central objective of this project is to integrate the work done by Eunjin (EJ) Jung et al. on Towards A Robust Detection of Malicious JavaScript (TARDIS) into the web browser via a Firefox add-on and to demonstrate the usability of our add-on in defending against such attacks. TARDIS uses statistical language modeling for an automatic feature extraction and combines it with structural features from an abstract syntax tree [1]. We have developed a Firefox add-on that is capable of extracting JavaScript code from the page visited and classifying the JavaScript code as either malicious or benign. We leverage the benefit of using a pre-compiled training model in JavaScript Object Notation (JSON). JSON is lightweight and does not consume much memory on a user's machine. Moreover, it stores the data as key-value pairs and easily maps to the data structures used in modern programming languages. The principle advantage of using a pre-compiled training model is better performance. Our model can achieve 98% accuracy on our sample dataset.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis advisor, Dr. Thomas Austin for his continuous support, valuable comments, and guidance throughout this project. I would also like to thank my committee members, Dr. Chris Pollett and Dr. Jon Pearce for their valuable time, and feedback. Very special thanks to Professor Eunjin (EJ) Jung, for helping me in understanding the project, and providing me the malicious dataset and the necessary platform to carry out the project execution.

Contents

Chapter

List of Tables

List of Figures

CHAPTER 1

Introduction

JavaScript and its frameworks are popular choice among web developers for building web pages. JavaScript can be placed in the HTML of web pages and can interface with the document object model of the page to provide extensive functionalities such as form validation, animation, asynchronous behavior, user activity tracking, interactivity, and more [9]. JavaScript is also used in server side code and in mobile applications by using cross-platform development tools such as Titanium and PhoneGap [10].

Since the release of JavaScript in 1995, many browsers and client-side security issues which have gained widespread attention [9]. JavaScript's capability to interact with the page's document object model makes it powerful, but at the same time, it also opens doors for attackers who can run malicious scripts on client computers by enabling a malicious agent to deliver the scripts over the internet. Malicious JavaScript has been listed in the Open Web Application Security Project (OWASP)'s 2013 Top 10 List of security issues [2]. Cross-site scripting has been listed as the third most widespread web application vulnerabilities on the Internet. Malicious JavaScript payload can be embedded into a legitimate website or web application by an attacker and can be executed on a client's machine. Several security measures have been taken to restrict the malicious code in order to access the client side sensitive information, the malicious JavaScript has access to the same objects as web pages and includes the user's cookies, sessions, etc. The malicious code can also redirect a user to an attacker's website and execute some malicious code without the user's permission, further advancing the attack to more severe ones.

One approach to solving this problem is to identify the pages that contain malicious scripts and either warn users before loading the page or block those scripts. The problem arises is how to distinguish malicious scripts from the benign ones accurately, as the dynamic nature of JavaScript makes it difficult to detect the exploit code. Moreover, attackers often use sophisticated obfuscation techniques that hide the malicious code and make detection complicated.

Recent work involves using machine learning techniques in combination with de-obfuscation and emulation technology [1]. Machine learning is used for feature extraction to identify the nature of the scripts. However, the malicious code keeps evolving, taking benefits of the dynamic feature of JavaScript though, they still need primitive JavaScript operations to be converted to clear text before execution [9]. A machine learning combined with de-obfuscation/emulation has proved to be advantageous, but they need a customized browser [1].

1.1 Our approach to the problem

Our approach is based on TARDIS [1]. TARDIS only requires the source code and does not utilize any de-obfuscation techniques on the original source code. TARDIS is simple yet achieves high accuracy compared to related research [1]. TARDIS uses machine learning techniques and robust features. Robust features are the features that can classify the malicious code with a high degree of accuracy. An attempt to conceal these features in the malicious code will require modifications in the malicious code generation algorithm, and to incorporate these modifications, an attacker will require additional resources.

The intuition on which TARDIS is based is the difference in the utilization of the JavaScript language for writing a benign program versus writing a malicious one.

An attacker writing malicious code attempts to conceal what the code is doing using various automated or manual procedures and involves the use of regular expressions, rules, or machine learning. A malicious program is likely to include more redundant parts as compared to a benign program. A benign, but poorly written JavaScript program may also include redundancy and inefficiency. However, an attacker's intention of bypassing the detection of the malicious code and the use of automation to generate obfuscated script tend to include much more redundancy and inefficiency as compared to a benign JavaScript program. TARDIS makes use of this difference. Furthermore, the features have been extended with a Statistical Language Model (SLM). SLM is termed as a probability distribution(s) of String S and estimates the frequency of a String S in a sentence [11]. SLM uses the general patterns in the language used in both malicious and benign JavaScript to classify benign and malicious JavaScript [1].

1.2 Firefox add-on

We have developed a Firefox add-on based on TARDIS. Once added to the browser, this add-on is capable of capturing the inline JavaScript from the current open tab. It then extracts the required features, performs analysis, and identifies the existence of an exploit. On detection of malicious JavaScript, the Firefox add-on alerts the user of the presence of an exploit in the current tab

Our Firefox add-on uses a precompiled training model in order to perform an efficient prediction. The precompiled training model has been stored in JSON. JSON is lightweight and allows a quick search. The training model has been computed over 15000 malicious and 30000 benign JavaScript files, and the model has been tested using more than 1000 malicious and 1000 benign JavaScript files. A 10-fold cross-validation has been performed in order to validate the model. The model tends to

reach 98% accuracy.

The remaining of the paper is organized as follows. In Chapter 2 we provide background information on SLM, XSS, and discuss TARDIS and other related work. Chapter 3 presents the Firefox add-on development and pre-compiled training model and similar security research by top companies and universities. In Chapter 4 we provide test results and accuracy of the training model, and Chapter 5 covers the conclusion. tradeoffs, and future work.

CHAPTER 2

Background

JavaScript is one of the primary languages in programming web technologies. It can interface with the document's object model (DOM) and provides different impressive functionality. Because of these features, JavaScript is extensively used on nearly every website, and all of the browsers allow JavaScript, as it helps in making the page dynamic and it keeps a user engaged.

JavaScript's capability of interacting with the DOM also grants it with the potential of injecting malicious code in the script dynamically. There has been various flavors and types of malicious JavaScript, and one of the most wicked ones is cross-site scripting (XSS).

2.1 Cross site scripting (XSS)

An XSS attack targets web applications that do not validate and sanitize user input such as form data, comments, etc. in a proper way; that enables attackers to inject malicious code into the web page. An attacker may insert a link to the third party malicious website into the benign web page. If a user visits such an infected page and clicks the link, the link will take the user to the malicious website and steal the user's cookies and other sensitive information stored in the browser. An attacker can use this information to impersonate that user. Attackers can also employ various kind of obfuscation technique to conceal the exploit in the link and makes it resemble like a legitimate link. There are commonly three types of XSS attacks: stored XSS, reflected XSS and DOM-based XSS [12].

2.1.1 Stored Cross Site Scripting

Stored cross-site scripting targets the websites that store the user input such as comments, form, etc. first in databases or the file system and later requested by the website users. If the input has not been sanitized or encoded and the data contains an attack, The user will receive the malicious script. This type of attack affects multiple users of the website [12].

Stored cross site scripting attack: attacker is storing malicious script to database using a form. The data is stored in the database without proper input validation and returned to the web user without output validation. A user clicks on the malicious link and the attacker hijacks the information stored in user's browser.

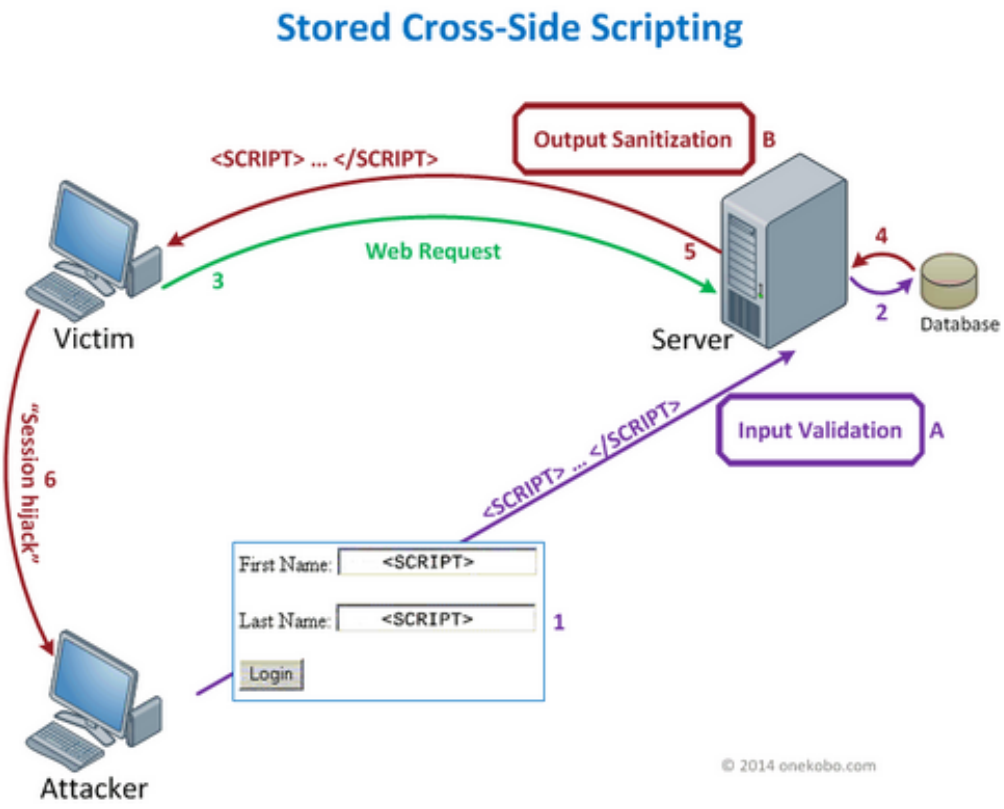


Figure 1: Stored cross-site scripting attack [14]

2.1.2 Reflected cross-site scripting

Reflected cross-site scripting targets the websites that reflect a user's input immediately to the web page. If not encoded, it may allow an attacker to introduce malicious code into the dynamic webpage. However, an attacker can only change his web page result, though the attacker can persuade a user to click on a link, which can lead that user to a malicious website [13].

Reflected cross site scripting attack: an attacker identifies a vulnerable website and inject malicious link. The attacker then convinces the user to click on the link using social engineering. The user clicks on the link and becomes victim of reflected XSS attack.

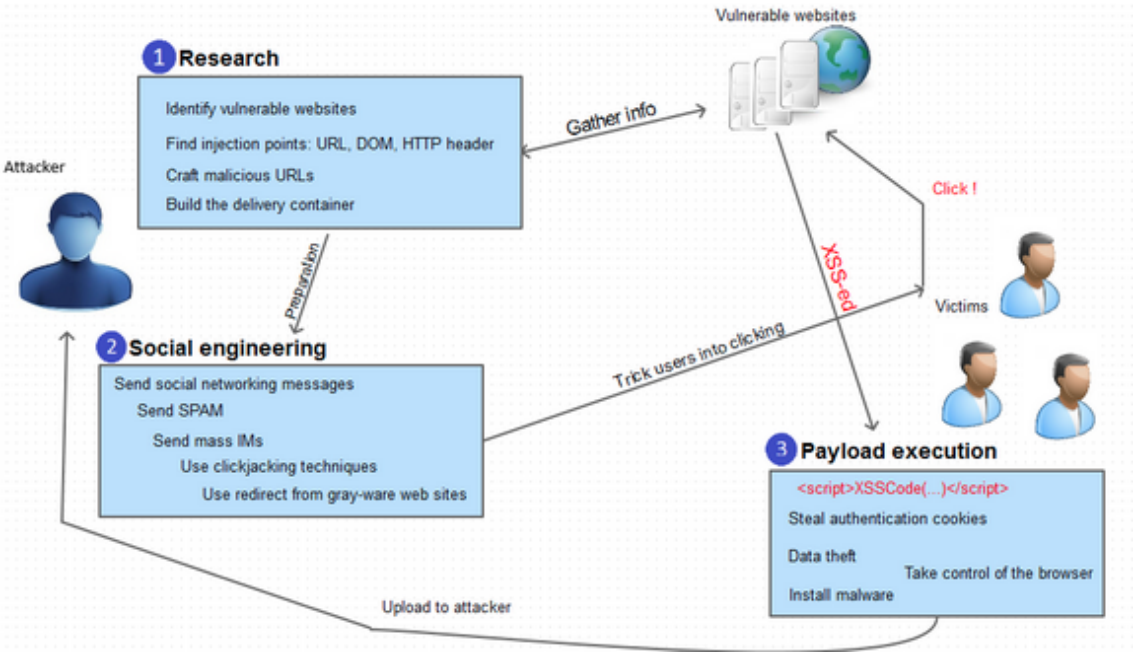


Figure 2: Reflected cross-site scripting attack [13]

2.1.3 DOM based XSS Attack

Every HTML page has an associated document object model (DOM) that consists of the HTML page objects. These objects represent the document properties. When a JavaScript within an HTML page executed, the browser provides the DOM of the HTML page to the script. A JavaScript can interact with the DOM and may perform an action based on the properties of the objects in DOM to make the page more interactive and dynamic. A DOM XSS attack targets the improper treatment of the data from its associated DOM in the HTML pages [20].

An example of DOM based XSS attack.

```
<html>
<head>
<title>Custom Dashboard </title>
...
</head>
Main Dashboard for
<script>
        var pos=document.URL.indexOf("context=")+8;
        document.write(document.URL.substring(pos,document.URL.length));
</script>
...
</html>
```

Figure 3: In this html page, JavaScript variable pos is set to the value of context field form the URL [20]

```
http://www.example.com/userdashboard.html?context=Mary
```

Figure 4: : User click on this URL which sets the variable pos to value of context i.e. Mary [20]

Example of the same URL with embedded malicious script.

```
http://www.example.com/userdashboard.html?context=<script>SomeFunction(somevariable)
http://www.example.com/userdashboard.html#context=<script>SomeFunction(somevariable)
```

Figure 5: An attacker embeds a malicious script as value of context field [20]

The user clicks on the above URL, which sends the request with the context value as malicious JavaScript. The browser builds the DOM of the web page after receiving the response from the server and sets the value of the property document.url to the value of the context. When the script gets executed it updates the raw HTML of the page with the malicious script and the malicious script now gets carried out by the browser resulting in the attack.

```
...
Main Dashboard for <script>SomeFunction(somevariable)</script>
...
```

Figure 6: HTML page with embedded malicious JavaScript [20]

2.2 Other variants of JavaScript Attack

Cross-site request forgery is also a standard JavaScript attack and has been listed as number five in the Top 10 web applications security risks by OWSAP 2013 [2]. Cross-site request forgery refers to sending malicious requests to an authorized user of websites that websites trusts. In cross-site request forgery, an attacker attempts to send a state change request such as a fund transfer or an email change. An attacker convinces an authorized user to execute unauthorized commands by use of social engineering tricks such as sending an email that looks authorized to the user. By

clicking on the link may submit that forged request if the user is already login to the website. A website has no way to know if the request is a legitimate one or a forged one as a website stored the login credentials and other sensitive information of the user in the cookies or session in the browser. That is why this attack is also known as session over-riding attack.

A legitimate request example:

Alice wants to transfer funds to Bob' account.

```
GET http://bank.com/transfer.do?acct=BOB&amount=100 HTTP/1.1
```

Figure 7: A legitimate fund transfer request to transfer money to Bob's account using GET request [20]

A malicious request

The attacker can change the value in GET request so that it transfers the fund to the attacker's account and tricks the victim using social engineering to click on the below link to transfer money to his account. The below forged requests can be by sent an email or can be injected in a website the user is most likely to visit while transferring funds.

```
http://bank.com/transfer.do?acct=MARIA&amount=100000
```

Figure 8: Malicious request forged by the attacker. Here name value is changed to MARIA form BOB and amount value is changed to 100000 form 100 [20]

2.3 Security measures adopted to prevent malicious JavaScript Attack

To avoid an attack, the following actions can be taken: escape and sanitize all the users input data, whitelist input validation, and employ content security policy using sandboxing. Modern web browsers are using the sandboxing and the same origin policy to prevent or restrain a JavaScript attack: [5]. Sandboxing limits the scope of a script, preventing the attacks from spreading system wide. The same origin policy prevents a script from one source to access resources from a different origin. However, attackers leverage the flaws in the websites and insecure practices and allowing them to circumvent the above two restrictions. The common defects and unsafe practices used by the attackers are vulnerable JavaScript inclusion and insecure JavaScript generation [15]. JavaScript inclusion injects the third domain JavaScript in the top level document by using the src attribute of a script tag and thus defy the purpose of same origin policy [15]. Attackers use eval() function for dynamic generation of malicious JavaScript code. According to research by [15], 66.4% of the website uses the insecure practice of JavaScript inclusion, and 74.9% uses dynamic JavaScript generation.

Modern approaches are using machine learning technology in combination with de-obfuscation/emulation for better performance and accuracy [1]. Machine learning can be used in analyzing and capturing the structural information of a malicious JavaScript program by extracting the abstract syntax tree, while emulation can be used to analyze the behavior of a malicious JavaScript program. Obtaining structural information for analysis is known as static analysis while using emulation to execute the exploit to examine and analyze the behavior and impact of an exploit is known as dynamic analysis. According to TARDIS [1], dynamic analysis tends to be more accurate than static analysis, but it has more performance overhead.

2.4 Static Analysis

Static analysis analyzes source code without executing it, and is commonly used as a technique for troubleshooting a computer program [26]. Static analysis helps in understanding the composition of a program. The static analysis examine weather a software application is correct and consistent in its organization and depiction [26]. It can be performed automatically using specific tools such as parsers, data flow analyzers, syntax analyzers, etc. Static analysis can also be followed by dynamic analysis for uncovering the subtle defects or vulnerabilities. Static and dynamic analysis together refers as glass box testing.

TARDIS is based on purely static analysis of malicious and benign JavaScript, and combines static analysis with SLM for robust feature extractions. In our project, we are using static analysis for analyzing the program syntax, and a JavaScript parser for capturing the abstract syntax tree, and examining the structure and usage of individual JavaScript statements, keywords, and reserved words. We are performing automatic static analysis by parsing the scripts in the add-on. A more detailed description of TARDIS is available in section 2.7.

2.5 Dynamic Analysis

Dynamic analysis involves examining source code by execution. It analyzes the action, impact, and behavior of software before and after the execution of the software in a controlled manner and environment. The execution of software can be carried out in either artificial or real application environment. Path testing and branch testing are two primary dynamic analysis techniques. Branch testing aims at traversing every branch of a program at least once while path testing attempts to exercise as many logical paths as possible [26].

Dynamic analysis and detection of a JavaScript exploit require a detection system that can observe and examine the execution of a JavaScript code during run-time. To capture this information, a JavaScript program is either executed in a sandbox environment or the detection system interacts with the JavaScript engine of the web browser. The detection system monitors and tracks the flow of the execution events, which result in modifications to the environment state [26].

2.6 Related Work

This section presents the recently advanced approaches in detecting and analyzing malicious JavaScript using machine learning technology. These approaches are either using static analysis, dynamic analysis or a combination of both.

2.6.1 JStill (Mostly Static Approach)

The JStill [6] approach is static. However, in conjunction with static analysis, JStill uses a lightweight runtime inspection, which helps in analyzing the essential characteristics of an obfuscated malicious program. JStill performs static analysis to capture the characteristics of an exploit. However, a static analysis alone may not be accurate due to the obscured nature of the malicious program. An obfuscated malicious program needs to be de-obfuscated before fulfilling its malicious intent and requires particular function invocations. JStill leverages this observation of function invocation to inspect the runtime behavior of obfuscated code. JStill examines the function invocation pattern by a malicious program using the browser's runtime operations and hence does not incur any extra performance overhead of dynamic analysis that requires executing an exploit in a controlled environment. JStill can be implemented in a browser. The average performance overhead of JStill is 4.9%. It shows

higher performance overhead i.e. $> 8\%$ for yahoo.com and sina.com.cn. JStill also tends to give a higher false positive rate for a benign obfuscated JavaScript program.

2.6.2 Zozzle: Fast and Precise In-Browser JavaScript Malware Detection

Zozzle [24] is a combination of both static and dynamic analysis. Zozzle mostly uses static analysis for better performance and high throughput. It also uses a component of dynamic analysis for better accuracy and the analysis of an obfuscated malicious JavaScript program. Static analysis of Zozzle uses Bayesian classification and it uses the JavaScript abstract syntax tree's hierarchical features to extract the essential predictive features and quick scanning. To handle the obfuscation, Zozzle uses a small runtime component. This component extracts and processes the JavaScript that is generated at runtime using `eval()`, `document.write()`, etc. It then sends this runtime generated code to its static analyzer right before the execution. Zozzle has a very high throughput as big as one megabyte of JavaScript code per second and an exceptionally low false positive rate of 0.0003%.

2.6.3 Cujo: efficient detection and prevention of drive-by-download attacks

Cujo [23] combined both static analysis and dynamic analysis for automatic detection and blocking of drive-by download attacks. Static analysis extracts lexical tokens representing reserved words, literals, and identifiers. The dynamic analysis uses a lightweight sandboxing environment that analyzes execution behaviors. Both the static and dynamic features are explained further using machine learning technique for robust detection of an exploit. Cujo can be embedded in a web proxy, and it tends to reach a very high accuracy of 94% in detecting an attack with a very low false positive rate. Cujo is a learning-based detection tool and uses the support vector

machine learning algorithm. In spite of high precision, the dynamic analysis part of Cujo incurs performance overhead and the run time of Cujo is 500 ms per web page.

2.6.4 EarlyBird: Early Detection of Malicious Behavior in JavaScript Code

EarlyBird [25] uses dynamic analysis to perform dynamic, efficient detection of an exploit. A dynamic analysis requires execution of an exploit which may also result in potential damage to the underlying system. EarlyBird attempts to prevent the severity of harm caused by the execution of a malicious script by detecting it in on early phase of execution. It uses a set of predefined events and JavaScript execution results in particular sequences of these events. These event tracking can be used for various features extractions. This sequence of events is then mapped to vector space and uses linear support vector machine algorithm for learning and detection to achieve better protection of the underlying system. EarlyBird restricts the amount of exploit code that gets through the execution by a factor of 2. EarlyBird makes use of support vector machine and can achieve a good performance of 93% with very low false positive.

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

Prophiler [16] uses static analysis for rapid detection of the presence of an exploit in a web page. Prophiler uses a JavaScript program to extract significant features from HTML content of a webpage. These features are then supplied to a machine learning technology. The primary purpose of Prophiler is to reduce the resources and cost of dynamic analysis tools for detection and analysis of a drive-by download attack. Dynamic analysis tools are capable of detecting a drive-by download attack precisely,

but they have costly analysis. This overhead is generally too costly for performing analysis on an extensive set of web pages. Prophiler is effective in reducing the load of dynamic analysis tools by 85%, but it still incurred 270 ms per page and has a 13.7% false positive rate.

2.6.6 Wepawet

Wepawet [8] uses an emulation techniques and combines it with anomaly detection for automatic identification of a drive-by download attack. Wepawet supplies the features of regular JavaScript to the machine learning classifier and uses emulation to detect the behavior of malicious anomalous JavaScript by analyzing it against previously verified features. Wepawet achieves a low false negative rate and no false positives on the data set tested.

2.6.7 PJScan: Static Detection of Malicious JavaScript-Bearing PDF Documents [4]

A pdf document is a commonly used file format, and they provide many features. Attackers have discovered a way to hide malicious scripts inside PDF files. PJScan uses static analysis on extracted JavaScript code to detect the JavaScript-bearing malicious PDF documents. PJScan incurs a significant low run-time overhead as compared to other previous work done that uses dynamic analysis approaches. PJScan can work efficiently on both known and unknown malicious JavaScript. PJScan utilized a lexical analysis approach and machine learning technology for automatic construction of the models, which can then be used to detect a pdf attack.

2.6.8 IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM

IceShield [7] performs in browser dynamic analysis and de-obfuscation to detect and mitigate a malicious JavaScript attack. IceShield is entirely based on dynamic analysis. It de-obfuscates the code first and then performs analysis on an exploit presented in clear text after de-obfuscation. IceShield primarily targets the types of attack that compromise the DOM and injects malicious code. IceShield makes use of a heuristic approach to discover an attacker from a benign user visiting and accessing the web page. IceShield can identify the fragment of the webpage that is malicious and modifies the page accordingly to block the attack. It is entirely implemented in JavaScript, and hence lightweight. It is also independent of a browser and can be applied in embedded browsers such as smartphone browsers. IceShield detection accuracy is 98%, and performance overhead is 12ms for a website and 80 ms for a smartphone.

Dynamic analysis provides better accuracy in detecting an exploit as compared to static analysis, but it incurs a performance overhead. Static analysis is faster than dynamic analysis, but not capable of detecting obfuscated malicious JavaScript efficiently. After examining the recent works done towards the detection of malicious JavaScript, we discover that most of the works are taking advantage of both approaches. They are trying to be mostly static to achieve the desired speed and implementing a lightweight dynamic analysis component for effectiveness without sacrificing performance.

A snapshot of a benign JavaScript program

```
"use strict";

var assert = require("assert");

var adapter = global.adapter;
var resolved = adapter.resolved;
var rejected = adapter.rejected;

var dummy = { dummy: "dummy" }; // we fulfill or reject with this when we don't intend to
test against it

describe("2.3.1: If `promise` and `x` refer to the same object, reject `promise` with a
`TypeError' as the reason.",
         function () {
    specify("via return from a fulfilled promise", function (done) {
        var promise = resolved(dummy).then(function () {
            return promise;
        });

        promise.then(null, function (reason) {
            assert(reason instanceof TypeError);
            done();
        });
    });

    specify("via return from a rejected promise", function (done) {
        var promise = rejected(dummy).then(null, function () {
            return promise;
        });

        promise.then(null, function (reason) {
            assert(reason instanceof TypeError);
            done();
        });
    });
});
```

Figure 9: A sample of benign script form test data set

A snapshot of a malicious JavaScript program

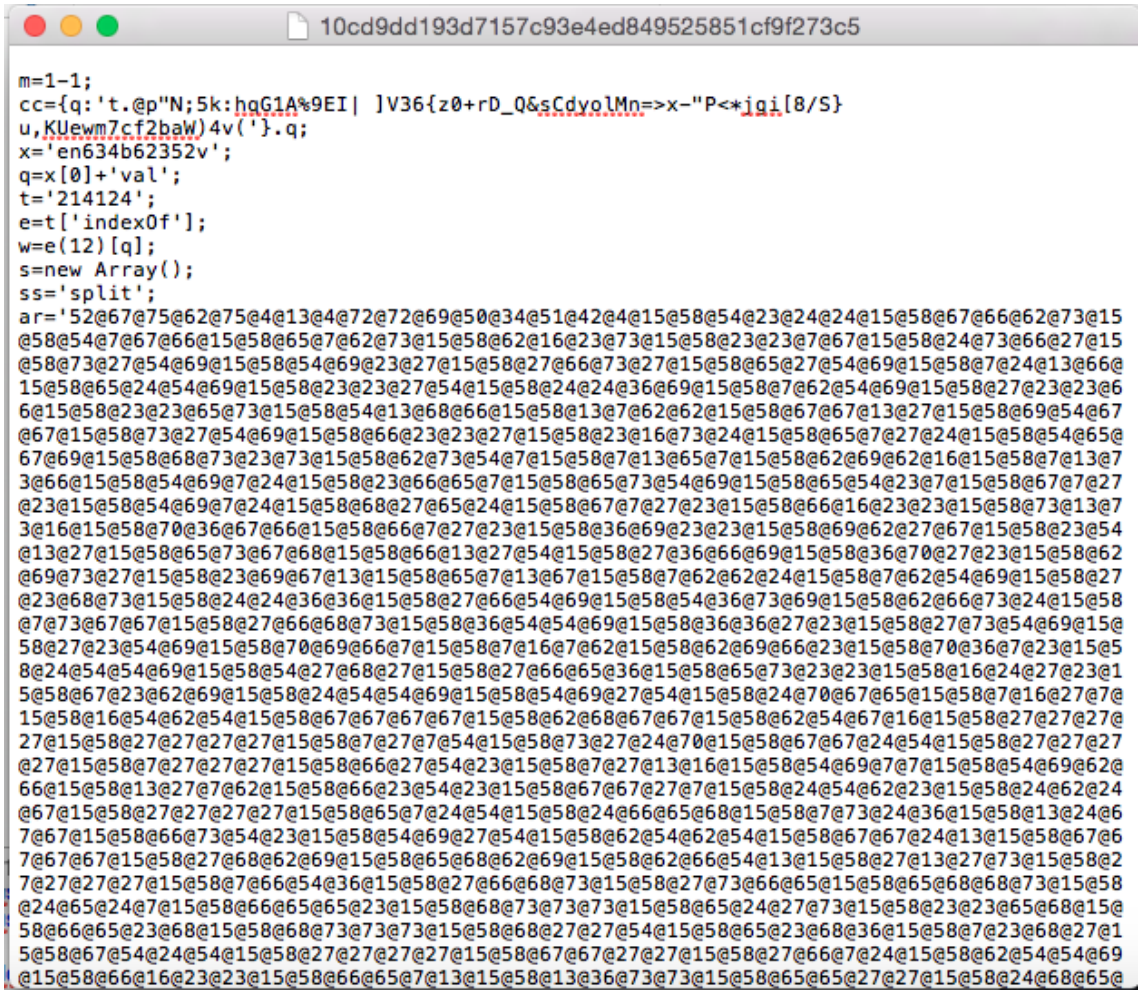


Figure 10: A sample of malicious script form test data set

A snapshot of a obfuscated JavaScript program

```
 function z8c231aa888(z14851c4b0f){return z14851c4b0f;}function z0ab1f0a49e(
z0721975593){document.write(z0721975593);}function zcd8c17c79d(z4716861143,
z500f443098,z9bc82e0042){z0ab1f0a49e(
"\x3c\x74\x61\x62\x6c\x65\x20\x62\x6f\x72\x64\x65\x72\x3d\x31\x3e");for(var
zd1ea46315e=(0x8e9+2039-0x10e0);zd1ea46315e<z4716861143.length;++zd1ea46315e){
var z708eb69ac7="\x3c\x74\x72\x3e";eval(z500f443098);z0ab1f0a49e(z708eb69ac7);
for(var z2d29194d43=(0x139b+2094-0x1bc9);z2d29194d43<z4716861143[zd1ea46315e].
length;++z2d29194d43){var z23b8891aeb="\x3c\x74\x64\x3e",z7f5411ee29=
"\x3c\x2f\x74\x64\x3e";eval(z9bc82e0042);z0ab1f0a49e(z23b8891aeb);z0ab1f0a49e(
z4716861143[zd1ea46315e][z2d29194d43]);z0ab1f0a49e(z7f5411ee29);}}z0ab1f0a49e(
"\x3c\x2f\x74\x61\x62\x6c\x65\x3e");}zcd8c17c79d([[(0x2d7+5314-0x1798),
(0xf7c+295-0x10a1),(0x900+1599-0xf3c)],[(0x1e8+1063-0x60b),(0xfc1+580-0x1200),
(0x1cf5+1843-0x2422)],[(0x9f9+4410-0x1b2c),(0x1c6+8452-0x22c2),
(0x28a+2774-0xd57)],[(0xcc0+2614-0x16ec),(0x7ee+1483-0xdae),(0xab2+6657-0x24a7)]
,[(0xa14+2966-0x159d),(0x63c+7549-0x23ab),(0x7e2+5079-0x1baa)],[
(0x14bc+296-0x15d4),(0x720+6090-0x1ed9),(0xfba+3045-0x1b8d)]],
"\x7a\x37\x30\x38\x65\x62\x36\x39\x61\x63\x37"+
"\x3d\x20\x27\x3c\x74\x72\x20\x73\x74\x79\x6c\x65\x3d\x22\x62\x61\x63\x6b\x67\x72\x6f\x75\x6e\x64\x3a\x20\x27\x20\x2
+"\x7a\x64\x31\x65\x61\x34\x36\x33\x31\x35\x65"+
"\x25\x32\x20\x3f\x20\x22\x72\x65\x64\x22\x20\x3a\x20\x22\x79\x65\x6c\x6c\x6f\x77\x22\x29\x20\x2b\x20\x27\x22\x3e\x2
,"");
```

Figure 11: A sample of obfuscated script form test data set

2.7 TARDIS

TARDIS (Towards Robust Detection of Malicious JavaScript) [1] developed by Professor E J Jung et al. at the University of San Francisco, is a completely static analysis tool. It only requires the source code of the exploit and hence does not require execution and thus avoids dynamic analysis performance overhead. Text based static analysis is not very useful in detecting obfuscated code as static analysis approaches tend to have a high false positive rate on minified, obfuscated benign scripts. To achieve optimal accuracy TARDIS has been supplemented with a powerful Statistical Language Model.

TARDIS's static analysis focuses on features that can differentiate between malicious and benign scripts based on their textual attributes. Analyzing textual attributes is purely static and does not require the execution of the source code. Some example of these textual attributes can be the use of whitespace, line breaks, the length of sentences, comments in a benign and malicious script, and the use of various keywords. These textual attributes can be used to discover a pattern in the way a malicious and benign JavaScript is written. These features alone are not sufficient for detecting a malicious code efficiently. An attacker may avoid detections by a slight change in their code generation algorithm, which requires analyzing more robust features incurring significant work on the part of the attacker in modifying their code generation algorithm to escape detection.

To achieve this requirement TARDIS makes use of a statistical language model for automated feature extraction by using a JavaScript parser and an abstract syntax tree in addition to the textual attributes features discussed in the previous section.

2.7.1 Abstract syntax tree

An abstract syntax tree defines the syntactical structure of a program by using nodes of a tree. An AST represents constants or variables as leaf nodes, and operators and statements as an inner node of the AST. Characteristic of an abstract syntax tree can be used to extract features that are difficult to be evaded by an attacker. Modification in the features of AST towards avoiding detection will require imitation of the AST of a benign code. A malicious code makes use of certain functions with higher frequency to carry out attacks such as string concatenation or `fromCharCode()` etc. Concealing the detection of these features by an AST will require the attacker to use a new algorithm to generate malicious code that avoids the textual attributes detection [1].

2.7.2 Statistical Language Modeling (SLM)

Statistical language modeling (SLM) [11] makes use of a statistical language model. A statistical Language model is defined as a probability distribution of a string (s) in a sentence [11]. The probability distribution of a string (s) represents the frequency of occurrence of (s) as a sentence. The most widely used SLM techniques are N-gram models and its variants [11].

TARDIS makes use of SLM for automatic feature extraction by employing a JavaScript parser. The JavaScript parser parses benign and malicious scripts and extracts essential features. These extracted features are then used to create SLM benign and SLM malicious training model that can be used to classify a benign or a malicious script.

2.7.3 TARDIS SLM model

The parser generates a collection of words based on certain delimiters after parsing a training corpus. These words then can be appended together and form an n-gram. N-gram represents a consecutive sequence of n words from a sentence. These n-grams constitute the features of the training model. The SLM training model describes the features as key-value pairs, where the key denotes a feature/n-gram and the values represents the probability of occurrence of that particular element in the model. This mapping of n-grams with probability forms the statistical model of TARDIS's static analysis technique. This mapping can then be used to compute the probability that a document belongs to a particular class (benign or malicious) [1].

TARDIS generates SLM models for benign and malicious scripts. SLM benign models are computed over benign scripts while the SLM malicious models are calculated using malicious scripts. While testing both the models are used to estimate the overall probability of a document belonging to either of the models. The model that gives the higher probability wins and the testing script is classified to the winning model.

TARDIS makes use of the following formula to estimate the likelihood of categorization of a script to either the benign or the malicious category.

2.7.4 N-grams SLM model

An n-gram model can have different forms, and each of these forms can be used in generating a model. Each of these models can provide different information and as well as the features and can have a different impact on the words and probability mapping, precision of the model. TARDIS experimented with models computed based on n-grams of size one, two, three, and four to tune the accuracy. N-grams of size

one considers each character as a feature while n-grams of size two joins together two consecutive characters. Similarly, a model based on n-grams of size three and four can be computed. N-grams model of size one tends to lose the surrounding context while n-grams model of a large size can provide too many surrounding contexts but less meaningful matches [1]. Mostly n-grams of size two or three provide meaning full match with adequate surrounding contexts. TARDIS built its training model for n-grams of size one to n-grams of size four and compute the accuracy of each of the model in order to identify which n-grams model provides better accuracy in terms of classification. TARDIS proposes the use of three categories of n-gram model. Each of them computes the benign and malicious training model for n-gram of size one, two, three, and four.

2.7.5 Character level n-grams

According to TARDIS, a character level n-grams model expresses the content of an input script rather than the composition of the input script. A character level n-grams model uses characters as tokens. It converts the input sequence to a collection of the characters and joins the consecutive characters to form different sizes of n-grams.

Given a sequence of input script as

```
var str = "javaScript"
```

An n-gram of size one will look like

```
['v', 'a', 'r', ' ', 's', 't', 'r', '=', '"', 'j', 'a', 'v', 'a',
'S', 'c', 'r', 'i', 'p', 't', '"']
```

An n-gram of size three will look like

```
['v', 'a', 'r'], ['a', 'r', ' '], [ 'r', ' ', 's'], [' ', 's', 't'],
['s', 't', 'r'], ['t', 'r', '='], ['r', '=', '"'], ['=', '"', 'J'],
['"', 'J', 'a'], ['a', 'v', 'a'], [''v', 'a', 's'], ['a', 'S', 'c'],
['a', 'S', 'c'], ['S', 'c', 'r'], ['c', 'r', 'i'], ['r', 'i', 'p'],
['i', 'p', 't'], ['p', 't', 'i']
```

A character level n-grams model can successfully extract useful predictive features such as JavaScript keywords, operators, and frequency of use of increment, decrement operators, etc. However, it is not very informative regarding the structure, and semantically meaningful input sequences such as function call as a character level n-grams model break down the function call into a list of characters.

2.7.6 Keyword Transformation

Keyword transformation n-grams model reserves all the JavaScript keywords as they are and uses them without breaking down into character tokens. It treats all the other input sequence the same as character level n-grams and calculates the model for different n-gram size. TARDIS uses a list of reserved JavaScript keywords to identify the keywords in an input script. Keyword transformation also does not count space character in the model generation.

Given a sequence of input script as

```
var s = 10;
```

Keyword transformation n-grams of size one will look like

```
['var', 's', '=', '1', '0', ';']
```

Keyword transformation n-grams of size three will look like

```
['var', 's', '='], ['i', '=', '1'], ['=', '1', '0'], ['1', '0', ';']
```

Here 'var' is a JavaScript keyword and hence, it is used as it is without breaking down into characters. Keyword transformation represents both the semantics and the content of a program. Keyword transformation can be used in extracting common programming language features such as variable assignments, which is helpful in classifying a benign script if it is not obfuscated [1]. However, it does not prove very beneficial in identifying malicious, obfuscated scripts [2].

2.7.7 Composite word-type transformation

Keyword transformation is not very accurate in analyzing obfuscated JavaScript. An obfuscated JavaScript program makes use of string encoding to conceal its payload. Keyword or character level conversion on an encoded string results in a substantial number of unique characters that do not present any significant information. To manage efficient detection of obfuscated malicious JavaScript, TARDIS is uses composite word type transformation. The composite word type transformation practices a predefined class based transformation. It assigns each token to a particular class and computes the probability model by computing the frequency of appearance of these classes in the model. Representing a program based on these classes reduces randomness in a program to more significant features. Commonly a program consists of digits, hexadecimal numbers, white spaces, punctuation, etc. Composite word type transformation provides a separate class for each type of element. Characters other than the above-defined classes are combined and interpreted as whole words.

Composite word type transformation n-grams of size one of 'var s = 10;'

```
['var', 'SPACE', 's',  'SPACE', 'PUNCTUATION', 'SPACE',
'DIGIT', 'PUNCTUATION']
```

Composite word type transformation n-grams of size three of 'var i = 3 ;'

```
['var', 'SPACE' , 's'], ['SPACE' , 's', 'SPACE'],
[ 's', 'SPACE','PUNCTUATION'], ['SPACE','PUNCTUATION','SPACE'],
['PUNCTUATION','SPACE', 'DIGIT'], ['SPACE', 'DIGIT', 'PUNCTUATION']
```

2.8 Malicious Probability Query Strategy

A composite word type transformation reduces randomness and uniqueness of an obfuscated JavaScript program and group together the unique characters using a predefined class. Probability model generation of an obfuscated script requires extra control over the method by which probability of a particular type of n-gram is estimated. TARDIS introduces an alphanumeric probability strategy for computation of malicious model. An alphanumeric probability strategy calculates the probability of string consists of only alphanumeric characters based on the following formula

$$(1/62)^n$$

where n is the length of the string. Here 62 is the sum of 26 upper case alphabets from A to Z, 26 lower case alphabets from a to z, and ten digits from 0 to 9.

TARDIS also performs smothering of the probability of an n-gram which is not present in the model to avoid setting the probability as zero.

CHAPTER 3

Firefox add-on Implementation

Firefox add-ons are a small piece of software that are used to extend and modify the installed version of Firefox by adding new features or functionality. An add-on can be used to change the theme or visual appearance of a website, add new features to the installed Firefox version, modify the user interface, add foreign language dictionaries, etc. Standard web technologies such as JavaScript, HTML and CSS are commonly used to develop a Firefox add-on [18].

3.1 Usability of our Firefox add-on

We have developed a Firefox add-on to integrate TARDIS with the web browser. It scans the JavaScript from the currently open tab and alerts the user to the presence of a malicious script, hence preventing the user from any further action in the currently open tab. The central purpose of developing a Firefox add-on is to show the usability and performance evaluation of TARDIS in the browser. The add-on is entirely developed in JavaScript and hence can be integrated with other analysis tools in JavaScript.

3.2 Developing a Firefox add-on

A Firefox add-on can be developed using either of the following two methods:

1. WebExtensions
2. Add-on SDK

3.3 WebExtensions

WebExtensions provide APIs for developing Firefox add-on, and is currently in the early state, but is considered to be the future of Firefox add-on development. According to [18], WebExtensions will become the standard by 2017. WebExtensions provide cross-browser compatibility, and the APIs are compatible with Google chrome and Opera's Extension API [18].

3.4 Add-on SDK

The add-on SDK method provides JavaScript APIs for Firefox add-on development and tools for creating, running, testing, and packaging them. Standard web technologies (JavaScript, CSS, HTML) are used in combination with the add-on SDK APIs. It requires Firefox version 38 or later [18].

We have developed our Firefox add-on using the add-on SDK. At the time we started development, add-on SDK was the most stable version available.

3.5 Firefox Add-on SDK installation and structure

The add-on SDK includes the jpm for initializing, running, testing, and packaging a Firefox add-on. jpm is based on Node.js. After installation, an empty add-on is initialized by running 'jpm init' inside an empty directory. The initial directory structure of a Firefox add-on looks like the following:

The figure shows the directory structure of the add-on. Here index.js is the entry point of the add-on and can be changed during the initial setup. Once the initial setup is done, Firefox add-on is developed using Add-on SDK's high-level and low-level APIs.

Figure 12: Initial directory structure of the Firefox add-on [18]

3.6 index.js

index.js 〉 No Selection

```
var self = require('sdk/self');
var tabs = require('sdk/tabs');
var buttons = require("sdk/ui/button/action");

var warning = require("sdk/panel").Panel({
  contentURL: self.data.url("text-warning.html")
});

buttons.ActionButton({
    id: "attach-script",
    label: "Attach the script",
    icon:{
      "16": "./image/malware-icon.png",
      "32": "./image/malware-icon.png"
    },
    onClick: runScript
});

function runScript() {
    console.time("index");
  var job = tabs.activeTab.attach({
    contentScriptFile: [self.data.url("models_50_wif/benign/InputBenignCountsA.js"),
                        self.data.url("models_50_wif/benign/InputBenignCountsAB.js"),
                        self.data.url("models_50_wif/benign/InputBenignTotal.js"),
                        self.data.url("models_50_wif/malicious/InputMaliciousTotal.js"),
                        self.data.url("models_50_wif/malicious/InputMaliciousCountsA.js"),
                        self.data.url("models_50_wif/malicious/InputMaliciousCountsAB.js"),
                        self.data.url("TestInput.js")]


  });

  job.port.on("script-response", function(response) {

    if (response == "malicious") {
        //code
    }
```

Figure 13: Index.js

Index.js is the entry point of our Firefox add-on. Index.js creates and adds a button to the current version of Firefox. On the onClick event of the add-on button, function runScript gets invoked. The runScript function is responsible for invoking the SLM_Script.js file and including the pre-build training models.

Index.js is our main add-on script. An add-on scripts can use the SDK's high-level and low-level APIs. But it does not get access to the web content directly. The add-on uses separate scripts known as content scripts to get access to the web content. To scan the JavaScript present on the page and detect malicious content, our add-on needs to access the web page content. Some of the SDK API's, like page-mod and tabs, provide necessary functions to load content-script. Here we are loading content scripts in our main SDK script using the tabs module's attach function. The attach function is using the contentScriptFile option to load content script as a file.

```
function runScript() {
    console.time("index");
  var job = tabs.activeTab.attach({
    contentScriptFile: [self.data.url("models_50_wif/benign/InputBenignCountsA.js"),
                        self.data.url("models_50_wif/benign/InputBenignCountsAB.js"),
                        self.data.url("models_50_wif/benign/InputBenignTotal.js"),
                        self.data.url("models_50_wif/malicious/InputMaliciousTotal.js"),
                        self.data.url("models_50_wif/malicious/InputMaliciousCountsA.js"),
                        self.data.url("models_50_wif/malicious/InputMaliciousCountsAB.js"),
                        self.data.url("TestInput.js")]


  });
```

Figure 14: function runscript

Tabs module is using attach () function to load the content scripts. Self.data.url(file_name) is pointing to the file inside data directory.

3.7 Content scripts

Content scripts can access web content, but like the main add-on scripts, content-scripts can't access the SDK's APIs. Content scripts are stored as separate files under

the data directory. The data directory is not created by default and needed to be added manually. We store all of our content scripts and a precompiled training model inside the data directory. The content script can communicate back its response to the add-on script using message passing APIs.

The message communication can be done using the property port of the global object self. The sender the of message calls `port.emit` to send message and the receiver calls port.on to receive the message.

3.8 Data Directory

The data directory contains the necessary content scripts that extracts the scripts from the web page of the current open tab and classify them as either benign or malicious category.

Figure 15: Add-on directory structure. Data directory contains models, image, and content scripts.

3.8.1 SLM_Script.js

SLM_Script.js is a content script. SLM_Script.js extracts the JavaScript from the web page and stores it in an array and then applies algorithm to automatically generate the n-gram based benign and malicious SLM models. The script can generate the following SLM models: character level n-grams of size three and four, keyword transformation n-grams of size three and four, and composite word type transformation n-grams of size three and four. These features are used by the precompiled benign and malicious training models to compute the overall probability of the script belonging to either of the models. The result is then passed to the add-on script index.js using `port.emit`.

```
1 self.port.emit("script-response", "compositeMaliciousScore = "
    + compositeMaliciousScore);
2 self.port.emit("script-response", "compositeBenignScore = " +
    compositeBenignScore);
```

Figure 16: Example of port.emit: SLM_scripts.js passing the final result to the index.js

3.8.2 Models

The Firefox add-on leverages the benefit of a pre-compiled training model for detection efficiency and better performance. The models directory inside the data directory holds all the precompiled training models required by the add-on. A script is tested on both the training model to detect the presence of malicious content. A precompiled model used within the Firefox add-on saves the overhead of sending and receiving a HTTP request to the server for the classification decision.

3.9 Pre-compiled training model

This section will present the detail discussion of the pre-compiled models we are utilizing for the add-on.

3.9.1 Types of pre-compiled models

We categorize all the training models to two categories: benign and malicious Each of the benign and malicious categories further contains models based on character level n-grams, keyword transformation n-grams, and composite word type transformation n-grams. We are computing n-grams models of each type of size three and four.

3.9.2 Character level n-gram model

To compute a character level n-gram model, a file is parsed and then converted to a list of characters, then consecutive characters are joined and stored as a key-value pair in JSON format. A key is the n-gram/feature and the value is the frequency of occurrence in the script. This type of model presents the content of the document more than the structure.

Figure 17: A snapshot of a pre-compiled malicious character level n-grams model of size four. Every key is four characters long

3.9.3 Keyword transformation

Keyword transformation parse the script and converted it into a list of characters, then join the consecutive characters to form n-grams. Keyword transformation is similar to character level n-grams, but in keyword transformation, reserved keywords are stored with the whole word as a single token. Keyword transformation preserves both the content and the semantics of a script.

Figure 18: a snapshot of a keyword transformation n-grams model. Reserved keyword such as length, constructor, and min appear as the whole word combined with the consecutive characters that are not part of the reserved keyword

3.9.4 Composite word type transformation

Composite word type transformation converts the sequence of characters into distinct classes. Here the following classes are used to represent characters: DIGIT, HEX, WHITESPACE, PUNCTUATION, and PERCENT. Characters other than these categories are joined and represent a single token. These classes and tokens are combined to form composite word type n-grams of size there and four. As discussed in the section 2.7.4, composite word type transformation reduces entropy in an obfuscated malicious program.

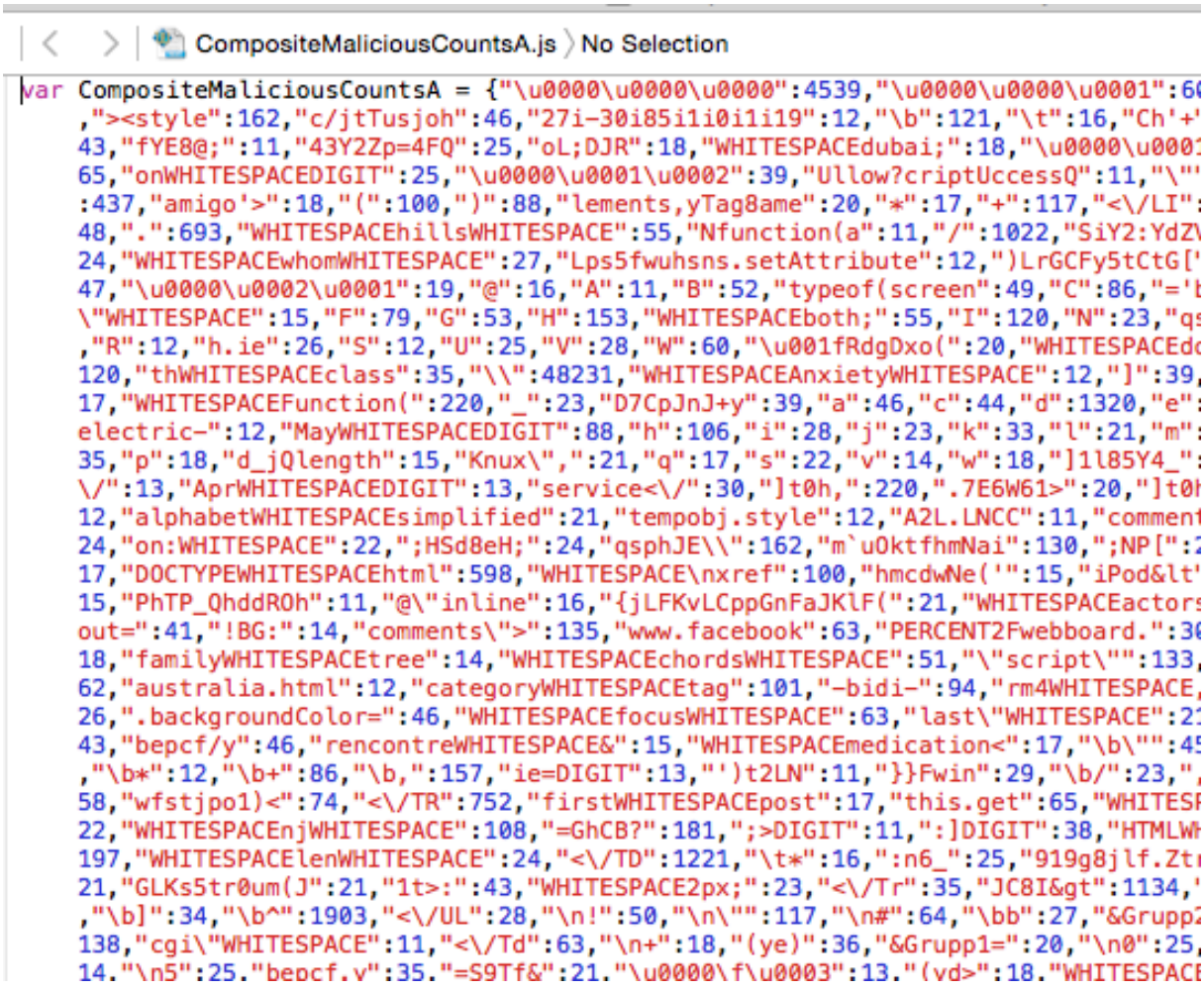


Figure 19: A snapshot of the malicious n-grams composite word type transformation model

3.9.5 Precompiled training models computation

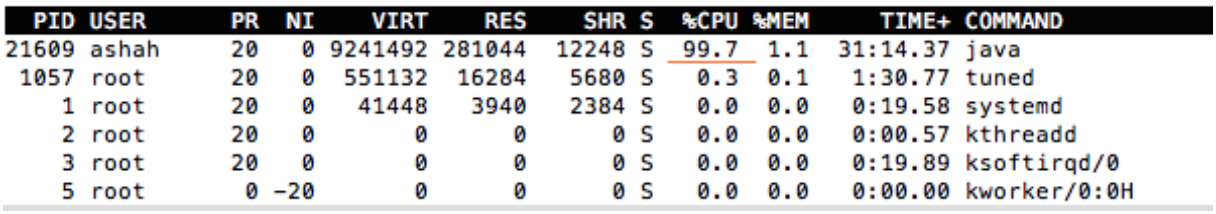
The models are computed using the TARDIS source program in Java. TARDIS is written in Java. The source code first computes the training model and uses the training model to test the JavaScript for malicious or benign categorization. We leverage this functionality and store the model generated by TARDIS persistently in JSON format. The primary reason behind storing a model in JSON format is that a JSON object is lightweight and portable. Storing a model in JSON with the add-on would not take much space in the browser and it can also provide a quick look up of key-value pair.

```
    JSONObject obj_benignCountsAB = new JSONObject();
    for(Map.Entry<TermSequence, Integer> entry :
benignModel.countsAB.entrySet()){
        String key = entry.getKey().toString();
        key = key.substring(1, key.length()-1).replace(", ",
"");
        int value = entry.getValue();

         try {
           //if(value > 10)
              obj_benignCountsAB.put(key, value);
        } catch (JSONException e) {
           e.printStackTrace();
        }
    }

    FileWriter file_benignCountsAB;
    try {
        file_benignCountsAB = new
FileWriter("KeywordBenignCountsAB_50.json");

file_benignCountsAB.write(obj_benignCountsAB.toString());
        file_benignCountsAB.flush();
        file_benignCountsAB.close();
    } catch (IOException e2) {
        e2.printStackTrace();
    }
```

Figure 20: Java code added for model computation

3.9.6 Problems faced during pre-compiled model generation and solution implementation

The model generation for large no of files is a computationally expensive process. For efficient processing and time reduction for model generation, we implemented a multithreading solution to the existing TARDIS model generation algorithm. The multithreading solution reduces execution time by roughly two-third.



PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
21609	ashah	20	0	9241492	281044	12248	S	99.7	1.1	31:14.37	java
1057	root	20	0	551132	16284	5680	S	0.3	0.1	1:30.77	tuned
1	root	20	0	41448	3940	2384	S	0.0	0.0	0:19.58	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.57	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:19.89	ksoftirqd/0
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H

Figure 21: output of top command before multithreading implementation shows % CPU utilization as 99.7%

```
       151335 forks
[ashah@leffe ~]$ vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 23314584   3128 631600    0    0     0     0    1    2  0  0 100  0  0
[ashah@leffe ~]$ vmstat 1 20
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 23312396   3128 631688    0    0     0     0    1    2  0  0 100  0  0
 1  0      0 23312496   3128 631696    0    0     0     0 1117  365  8  0 92  0  0
 1  0      0 23312592   3128 631744    0    0     0     0 1088  314  8  0 92  0  0
 1  0      0 23312592   3128 631744    0    0     0     1 1080  291  8  0 92  0  0
 1  0      0 23312592   3128 631744    0    0     0     0 1078  289  8  0 92  0  0
 1  0      0 23312592   3128 631744    0    0     0     0 1075  271  8  0 92  0  0
 1  0      0 23312592   3128 631744    0    0     0     0 1085  301  8  0 92  0  0
 1  0      0 23312592   3128 631744    0    0     0     0 1072  270  8  0 92  0  0
 1  0      0 23312592   3128 631744    0    0     0     0 1073  284  8  0 92  0  0
 1  0      0 23312716   3128 631744    0    0     0     0 1070  276  8  0 92  0  0
 1  0      0 23312716   3128 631744    0    0     0     0 1116  357  8  0 92  0  0
 2  0      0 23312644   3128 631788    0    0     0     0 1083  318  8  0 92  0  0
 1  0      0 23312676   3128 631800    0    0     0     0 1088  310  8  0 92  0  0
 1  0      0 23312676   3128 631800    0    0     0     1 1071  285  8  0 92  0  0
 1  0      0 23312676   3128 631800    0    0     0     0 1096  306  8  0 92  0  0
 1  0      0 23312676   3128 631800    0    0     0     0 1072  303  8  0 92  0  0
 1  0      0 23312676   3128 631800    0    0     0     0 1113  318  9  0 92  0  0
 1  0      0 23312644   3128 631812    0    0     0     0 1071  283  8  0 92  0  0
 1  0      0 23312644   3128 631812    0    0     0     0 1091  301  8  0 92  0  0
 0  1      0 23312644   3128 631816    0    0     0     0 1049  348  8  0 92  0  0
```

Figure 22: CPU idle time before multithreading implementation = 92

CPU utilization percentage and idle time after multithreading implementation.

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 5505 ashah     20   0 9507744 270204  12280 S 346.2  1.1   0:24.56 java
  697 root      20   0       0      0      0 S  17.6  0.0 281:22.91 md1_raid5
 3259 root      25   5       0      0      0 D   3.7  0.0  10:04.34 md1_resync
  668 root       0 -20       0      0      0 S   0.7  0.0   8:07.50 kworker/5:1H
    1 root      20   0   41532   4032   2384 S   0.0  0.0   0:25.58 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.84 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:25.37 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    7 root      rt   0       0      0      0 S   0.0  0.0   0:00.24 migration/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcuob/0
   10 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcuob/1
```

Figure 23: output of top command after multithreading implementation shows % CPU utilization as 346.2%

```
[ashah@leffe input]$ vmstat 1 10
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 5  0      0 20638144   3128 3290476    0    0     0     0    0    1  3  0 97  0  0
 4  0      0 20637880   3128 3290476    0    0     0     0 4245 28358 22  2 76  0  0
 5  0      0 20637880   3128 3290476    0    0     0     0 4600 29676 22  2 76  0  0
 4  0      0 20637880   3128 3290476    0    0     0     0 4509 25998 22  2 77  0  0
 3  0      0 20611008   3128 3290476    0    0     0     0 4899 22013 23  4 73  0  0
 2  0      0 20611008   3128 3290476    0    0     0     0 3641 28899 18  2 80  0  0
 3  0      0 20611008   3128 3290476    0    0     0     0 3415 30799 16  2 82  0  0
 3  0      0 20611008   3128 3290476    0    0     0     0 3663 31191 17  2 81  0  0
```

Figure 24: CPU idle time after multithreading implementation = 76

3.10 Firefox add-on implementation

After the installation, the Firefox add-on appears in the browser toolbar on the right side. A user can click on the add-on to perform malicious scripts detection. The add-on extracts the JavaScript from the page and parses the script. After parsing, the add-on calculates the probability score of the script and classify it either benign and malicious based on the score. The add-on then passes the result to console using `port.emit`.

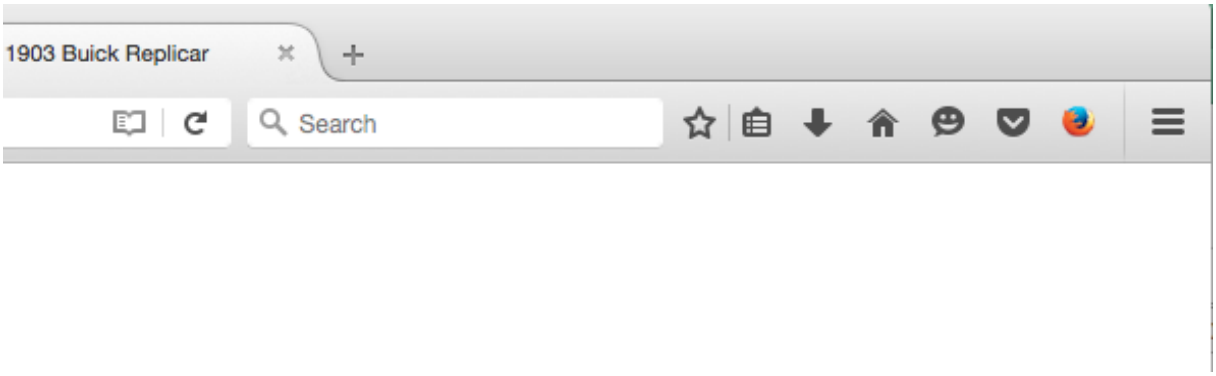


Figure 25: Firefox add-on in the browser

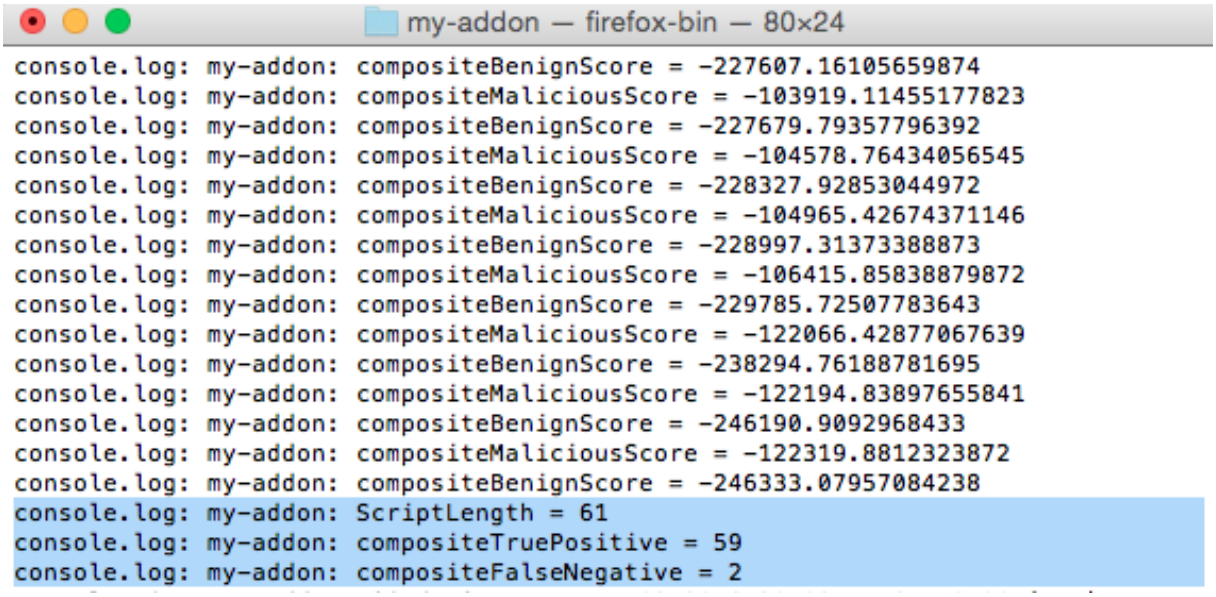


Figure 26: Firefox add-on detection result in the console

3.11 Result Computation

The add-on computes the overall probability of a script over the benign and malicious model. For each n-gram the mode looks for the frequency value in n-grams of size three model and n-grams of size four JSON model. The model then computes the overall probability of the script for both the benign and malicious models using the formula [1]

$$Probability = probability + math.log(pAB/pA)$$

$$pAB = (frequency\ of\ n-grams\ of\ size\ four)/(total\ no\ of\ words)$$

$$pA = (frequency\ of\ the\ n-grams\ of\ size\ three)/(total\ no\ of\ words)$$

CHAPTER 4

Testing

4.1 Dataset

We have obtained dataset for our model computation from different sources. We have collected a significant amount of both malicious and benign scripts to train our model, and we have made the effort to include various types of malicious scripts such as redirection, obfuscation, etc. For benign scripts set, we have also considered minified obfuscated benign scripts.

4.1.1 Malicious scripts

we have collected over 50000 of malicious scripts from EJ Jung et al. and the research team from the University of San Francisco. To train our model, we are utilizing 15000 of malicious datasets of a size of total 200 megabytes. Half of the malicious scripts is of type redirection, and other half represents all the other forms of attack.

4.1.2 Benign Scripts

We have collected the benign scripts from various resources on the internet. We have obtained over 27000 of benign files of total size equal to 200 megabytes. These files represent both clear and obfuscated benign scripts. Most of the benign files are from the JavaScript libraries such as React.js, MooTools, JQuery, D3.js, Processing.js, etc.

4.1.3 Problems with the scripts

In out dataset, it has been observed that malicious script size is commonly bigger than the benign script size. To match the different size, we are using maximum 15000 files for malicious model computation and over 30000 for benign model computation. We have also made sure that that both the models are of equal size to avoid over-fitting.

4.2 Training models

We are testing the add-on for various size of the models. We have observed that while calculating models if we optimize the model and don't consider the n-grams with the frequency less than 10, the model size gets reduced significantly. However, this reduction in size may incur a loss in accuracy. We have tested the add-on for both optimized and non-optimized version of each type of transformation. We are capturing accuracy and detection time with the different size of the models of each category to identify the maximum size of the training model that the add-on can utilize without sacrificing the performance.

We have computed benign and malicious models for a total file size of 50 mega-bytes for all the three kinds of transformation: character level n-gram, keyword transformation, and composite word type transformation. A detailed description of these transformation can be found in section 2.7.4.

4.3 Evaluation of n-grams models

We have evaluated all the three models for accuracy and performance. This section describes in details the performance and accuracy trade-off in between the models.

4.3.1 Evaluation of optimized and non-optimized models of character level n-grams.

We have evaluated all the three models for accuracy and performance. This section describes in details the performance and accuracy trade-off in between the models.

Table 1: Performance comparison of optimized and non-optimized character level n-grams model.

Model	**size(mb)**	**Scripts**	**TP**	**FP**	**TN**	**FN**	**Accuracy**	**Detection time/-file**
Character(opt)	8	3574	1299	164	1630	481	82%	150ms
Character(non-opt)	33.6	3618	1430	170	1624	394	85%	507ms

We observe that the no of scripts computed are changing with the change in models. If the model calculates the probability of a script as 0, it does not take that particular script under consideration.

Table 2: Accuracy and precision evaluation of optimized and non-optimized character level n-grams model.

Model	**Accuracy**	**Malicious Precision**	**Benign Precision**
Character(opt)	82%	41%	59%
Character(non-opt)	85%	44%	56%

A non-optimized character level n-grams model has 85% accuracy on the sample dataset as compared to the optimized character level n-grams model. However, there is a trade off in between benign and malicious precision and size of the model. A non-optimized model can identify a malicious script with high precision but the model size is 4 times bigger as compared to optimized one.

4.3.2 Keyword transformation

Table 3: Performance comparison of optimized and non-optimized Keyword transformation n-grams model.

Model	**size(mb)**	**Scripts**	**TP**	**FP**	**TN**	**FN**	**Accuracy**	**Detection time**
Keyword(opt)	7.31	3513	638	0	1767	1108	68.4%	5105ms
Keyword(non-opt)	31.6	3476	665	0	1767	1044	70%	2217ms

Table 4: Accuracy and precision evaluation of optimized and non-optimized keyword transformation level n-grams model.

Model	**Accuracy**	**Malicious Precision**	**Benign Precision**
Keyword(opt)	68.4%	18%	82%
Keyword(non-opt)	70%	19.1%	80.9%

The keyword transformation models achieve notable low accuracy as compared to the other two models. The accuracy improves for the non-optimized version of the model. We also observe that the models's malicious precision is very high as compared to model's benign precision. The low accuracy and precision of the model can be attributed to the large difference in the total no of words in keyword transformation benign models and the keyword transformation malicious model. The reason of such big difference can be that a malicious obfuscated models may have more random strings and less reserved keywords as compared to the benign model of same size.

```
1  var KeywordBenignTotal = {"totalWordsA":28340994,"totalWordsAB":28340994}
```

Figure 27: Total number of words in benign keyword transform model

```
var KeywordMaliciousTotal = {"totalWordsA":39453710,"totalWordsAB":39453710}
```

Figure 28: Total number of words in malicious keyword transform model

4.3.3 Composite word type transformation

Table 5: Evaluation of composite word type transformation n-grams model.

Model	size(mb)	Scripts	TP	FP	TN	FN	Accuracy	Detection time
Composite(opt)	11	3544	1687	9	1758	90	97.2%	1000ms

Table 6: Accuracy and precision evaluation of composite word type transformation n-grams model.

Model	Accuracy	Malicious Precision	Benign Precision
Composite(opt)	97.2%	47.9%	52.1%

The composite word type transformation model provides a very good accuracy of 98.7%. compared to other two types n-grams model with reasonable performance. The good performance of composite word type n-grams model is due to the reason that it represents the characters in the script to the set of classes. A malicious obfuscated script may contain random strings which may not represent the semantics of a script. However, if these random strings are converted to some set of particular words, then it provides more meaning to the script.

Here we have not considered the non-optimized model of the composite word type transformation due to its large size i.e. 174.9 megabytes. A model of such a large size is not optimal for a Firefox add-on in terms of performance and space complexity. The composite word type transformation also achieves good malicious and benign precision.

4.4 Model comparisons regarding accuracy and detection time on sample data set.

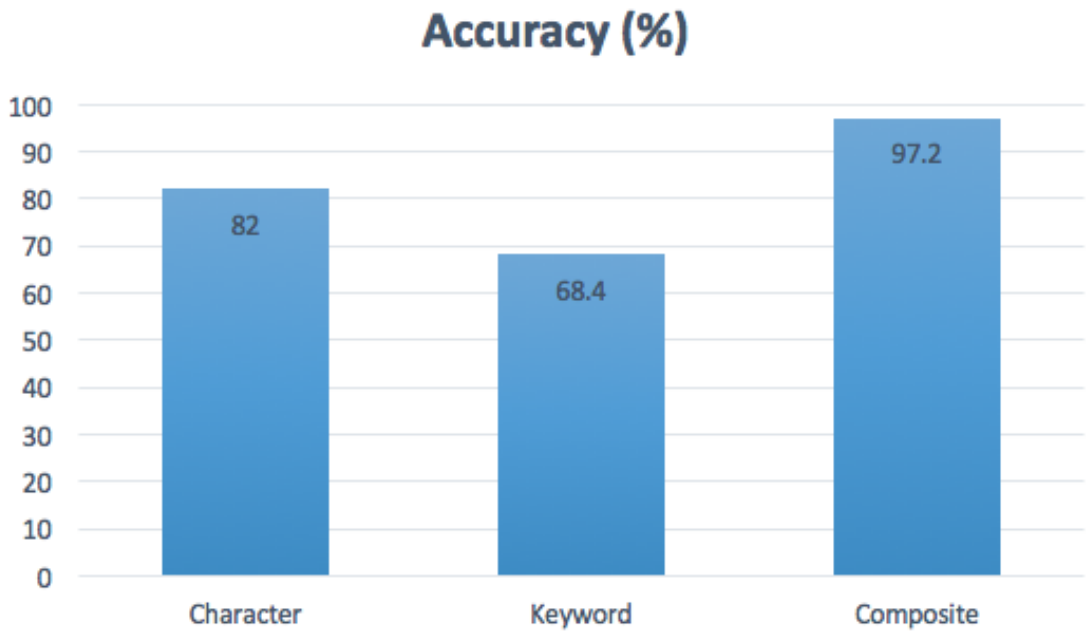


Figure 29: Accuracy comparison of all the three models

The composite word type transformation n-grams model achieves the highest accuracy compared to character level n-grams and keyword transformation n-grams models on the sample dataset.

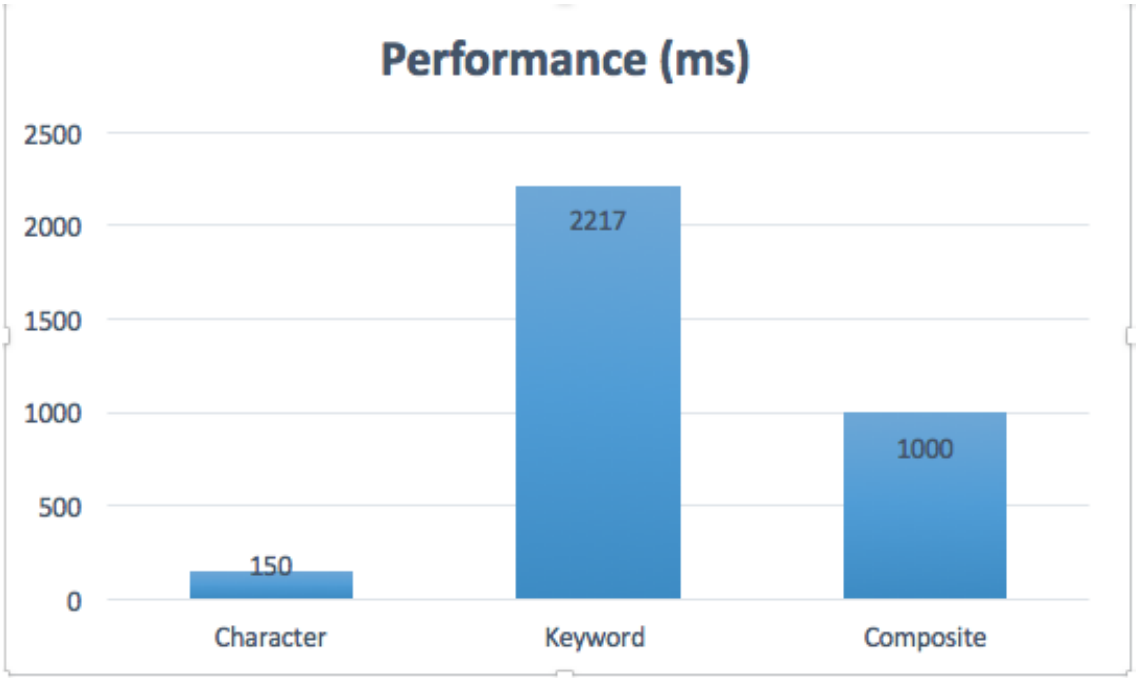


Figure 30: Performance comparison of all the three models

A character level n-grams model provides the lowest detection time in comparison to the other two models. Keyword transformation performs worst.

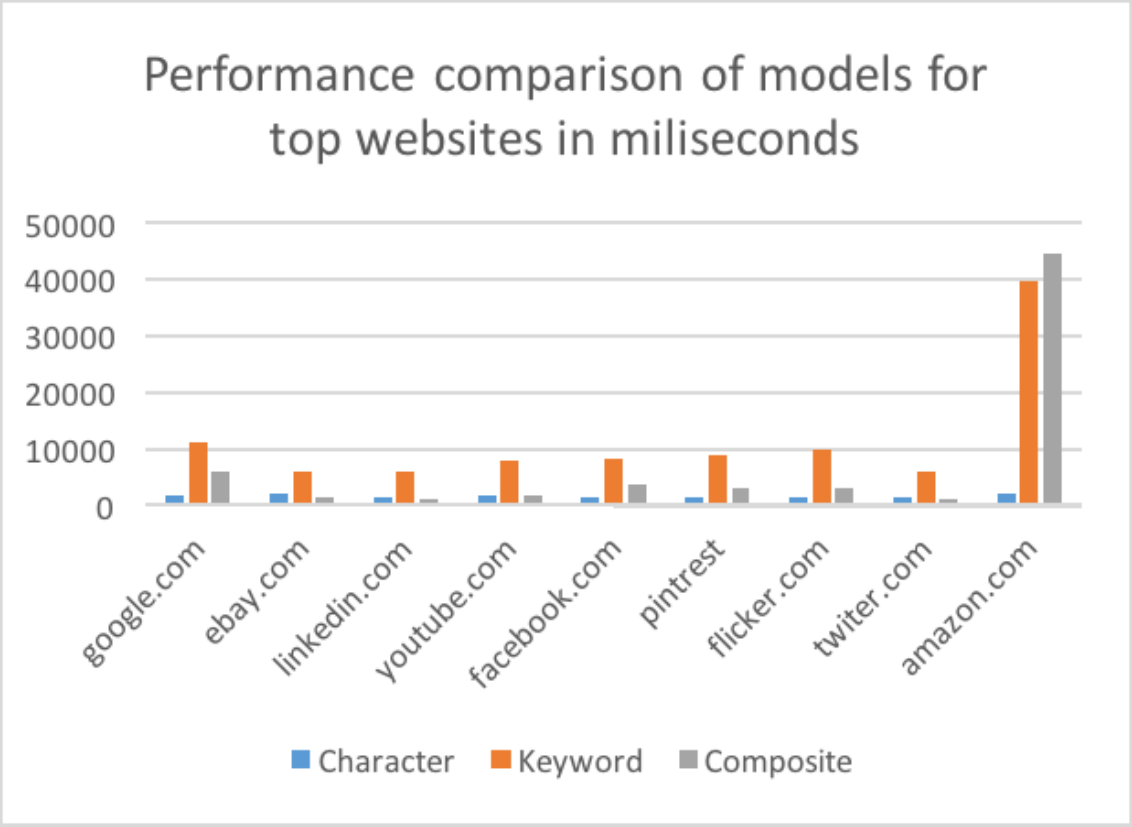


Figure 31: Performance comparisons of all the three models in real word scenario for the top websites

The composite word type transformation n-grams model provides an average detection time of 2.7 second with very good accuracy of 98%. The character level n-grams model performs best in terms of detection time. However, the character level n-grams model does not achieve the best accuracy. Keyword transformation performs worst in terms of accuracy and detection time. We also observe that both the keyword and composite transformation give the worst performance for amazon.com.

CHAPTER 5

Summary

Our experiments shows that our Firefox add-on achieves a maximum accuracy of 97.2% with the average detection time 1 s. We also observe that the composite word type transformation has better accuracy than character level and keyword transformation n-grams models. However, there is a trade-off between the accuracy and the performance in between the character level n-grams model and composite transformation model. The composite word type transformation model achieves very high accuracy but, it also requires high detection time for certain websites. Similarly, the character level transformation provides accuracy of 85% but it achieves high performance. The keyword transformation model performs the worst compared to the other two models with regard to both the accuracy and detection time. To the best of our knowledge, our add-on is the only one of its kind that is using a precompiled training model stored in JSON format within add-on. The add-on achieves similar accuracy to TARDIS. We have also observed that while computing the model and converting to JSON, we are losing certain encoded data which may have a significant effect on the accuracy of the model.

Given the size of the dataset and training model, our add-on achieves an excellent performance. However, attackers continuously find new and evolved method to perform attacks. To incorporate the new features required for detection of the evolving attacks, we need to keep on updating our precompiled model on a timely basis. To do this, one could have a server which continuously collects new data set and computes the new model and then updates the old one. In the current set-up, the model computation is a computationally expensive- process. To improve this process,

we can leverage map-reduce or other similar technology that can handle large files in small time.

LIST OF REFERENCES

[1] Peter Likarish, Eunjin (EJ) Jung, Chris Boyce. *Towards a Robust Detection of Malicious JavaScript(TARDIS)*, Unpublished paper

[2] The Open Web Application Security Project(OWASP). The owasp top ten project. `https://www.owasp.org/index.php/Category: OWASP_Top_Ten_Project./`

[3] Peter Likarish, Eunjin (EJ) Jung, Insoon Jo. Obfuscated Malicious JavaScript Detection using Classification Techniques, *second edition, Prentice Hall,* 1991.

[4] Pavel Laskov and Nedim Srndic. Static Detection of Malicious-JavaScript-bearing PDF Documents. *In Proceedings of the 27th Annual Computer Security Applications Conference.* ACM, 2011.

[5] Same-origin policy. Web. Retrieved December 2015, from `https://developer.mozilla.org/en- US/docs/Web/Security/Same-origin_policy`

[6] Wei Xu, Fangfang Zhang, Scencum Zhu. JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code. *In Proceedings of the third ACM conference on Data and application security and privacy. ACM, 2013.*

[7] Mario Heiderich, Tilman Frosch and Thorsten Holz. IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM. *Recent Advances in Intrusion Detection Volume 6961 of the series Lecture Notes in Computer Science.* pp 281-300.

[8] Phu H. Phung, Lieven Desmet. A Two-tier sandbox architecture for untrusted JavaScript. *In Proceedings of the Workshop on JavaScript Tools.* Pages 1-10. ACM, 2012.

[9] Fergal Glynn. JavaScript Security. Web. Retrieved March 2015, from `http://www.veracode.com/security/javascript-security`

[10] Kevin Whinnnery. Comparing Titanium and PhoneGap. (May 12, 2012). Web. Retrieved March 2015, from `http://www.appcelerator.com/blog/2012/05/comparing-titanium-and-phonegap/`

[11] What is Statistical Language Modeling. Web. Retrieved April 2015, from `http://homepages.inf.ed.ac.uk/lzhang10/slm.html`

[12] Cross Site Scripting. Retrieved April 2015, Web. Retrieved March 2015, from `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`

[13] Non- Persistent cross site scripting. Web. Retrieved April 2015, from
http://www.acunetix.com/blog/articles/non-persistent-xss/

[14] Client-Side Attacks. Web. Retrieved April 2015, from
http://neokobo.blogspot.com/2014/01/3218-client-side-attacks.html

[15] Chuan Yue, Haining Wang. Characterizing Insecure JavaScript Practices on the Web. *In proceedings of the 18th international conference on World wide web.* pages 961-970. ACM 2009.

[16] Davide Canali, Marco Cova, Giovanni Vigna, Christopher Kruegel. Prophiler: A Fast Filter for the Large-Scale Detection of Malicious Web Pages. *In proceedings of the 20th international conference on World wide web.* pages 197-206. ACM 2011

[17] Obfuscation of client side JavaScript. Web. Retrieved April 2015, from
http://stunnix.com/prod/jo/sample.shtml

[18] Firefox Add-ons SDK. Web. Retrieved April 2015, from
https://developer.mozilla.org/en-US/Add-ons

[19] Cross-Site Request Forgery (CSRF). Web. Retrieved April 2015, from
https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

[20] DOM-based Cross-Site Scripting(XSS) explained. Web. Retrieved April 2015, from
http://www.acunetix.com/blog/articles/dom-xss-explained/

[21] ChengXiang Zhai. Statistical Language Models for Information Retrieval A Critical Review. Web. Retrieved March 2016. From
http://sifaka.cs.uiuc.edu/czhai/pub/slmir-now.pdf

[22] M. Stamp, *Information Security: Principles and Practice*, second edition, Wiley, 2011

[23] Konrad Rieck, Tammo Krueger, Andreas Dewald. Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. *In Proceedings of the 26th Annual Computer Security Applications Conference.* Pages 31-39. ACM, 2010

[24] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, Christian Seifert. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. *In Proceedings of the 20th USENIX conference on Security.* Pages 3-3. USENIX Association 2011

[25] Kristof Schutt, Marius Kloft, Alexander Bikadorov, Konrad Rieck. Early Detection of Malicious Behavior in JavaScript Code. *In Proceedings of the 5th ACM workshop on Security and artificial intelligence.* Pages 15-24. ACM, 2012

[26] static analysis (static code analysis). Web. Retrieved April 2015, from http://searchwindevelopment.techtarget.com/definition/static-analysis

[27] Glass box testing. Web. Retrieved April 2015, from http://www.issco.unige.ch/en/research/projects/ewg95/node81.html