

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

© 2016

Anumeha Shah

ALL RIGHTS RESERVED

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 Mathematics

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 most advantageous 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 deobfuscation 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 primary objective of this project is to integrate the previous work 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 combines statistical language modeling for automatic feature extraction with structural features from an abstract syntax tree. 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 want to me, myself, and I.

Contents

Chapter

1	Introduction	1
1.1	Our approach to the problem	2
1.2	Firefox add-on	3
2	Background	5
2.1	Cross site scripting (XSS)	5
2.1.1	Stored Cross Site Scripting	6
2.1.2	Reflected cross-site scripting	7
2.1.3	DOM based XSS Attack	8
2.2	Other variants of JavaScript Attack	9
2.3	Security measures adopted to prevent malicious JavaScript Attack	11
2.4	Static Analysis	12
2.5	Dynamic Analysis	12
2.6	Related Work	13
2.6.1	JStill (Mostly Static Approach) [?]	13
2.6.2	Zozzle: Fast and Precise In-Browser JavaScript Malware Detection [?]	14
2.6.3	Cujo: efficient detection and prevention of drive-by- download attacks [?]	14
2.6.4	IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM [?]	15
2.6.5	EarlyBird: Early Detection of Malicious Behavior in JavaS- cript Code [?]	15

2.6.6	Wepawet [?]	16
2.6.7	PJScan: Static Detection of Malicious JavaScript-Bearing PDF Documents [?]	16
2.6.8	Prophiler: A Fast Filter for the Large-Scale Detection of Malicious Web Pages [?]	17
2.7	TARDIS	21
2.7.1	Abstract syntax tree	22
2.7.2	Statistical Language Modeling (SLM) [?]	22
2.7.3	TARDIS SLM model	23
2.7.4	N-grams SLM model	23
2.7.5	Character level n-grams	24
2.7.6	Keyword Transformation	25
2.7.7	Composite word-type transformation	26
2.8	Malicious Probability Query Strategy	27
3	Firefox add-on Implementation	28
3.1	Usability of our Firefox add-on	28
3.2	Developing a Firefox add-on	28
3.3	WebExtensions	29
3.4	Add-on SDK	29
3.5	Firefox Add-on SDK installation and structure	29
3.6	index.js	30
3.7	Content scripts	31
3.8	Data Directory	32
3.8.1	SLM_Script.js	34

3.8.2	Models	34
3.9	Pre-compiled training model	35
3.9.1	Types of pre-compiled models	35
3.9.2	Character level n-gram model	36
3.9.3	Keyword transformation	37
3.9.4	Composite word type transformation	38
3.9.5	Precompiled training models computation	39
3.9.6	Problems faced during pre-compiled model generation and solution implementation	40
3.10	Result Computation	42
4	Testing	43
4.1	Dataset	43
4.1.1	Malicious scripts	43
4.1.2	Benign Scripts	43
4.1.3	Problems with the scripts	44
4.2	Training models	44
4.3	Evaluation of n-grams models	44
 APPENDIX		
A	Zorak Likes Beans	45
A.1	Oh Yes He Does	45
A.2	Really	45
B	Everybody Wants to Be Space Ghost	46

List of Figures

1	Stored cross-site scripting attack	6
2	Reflected cross-site scripting attack	7
3	DOM based cross-site scripting attack	8
4	DOM based cross-site scripting attack example	8
5	DOM based cross-site scripting attack example	9
6	HTML page with embedded malicious JavaScript	9
7	Cross-site request forgery attack	10
8	Malicious request forged by the attacker	10
9	Benign script sample	18
10	Malicious script sample	19
11	Obfuscated script sample	20
12	Initial directory structure of the Firefox add-on	30
13	Index.js	30
14	function runScript.js	31
15	Add-on directory structure	33
16	Example of port.emit	34
17	A snapshot of a pre-compiled malicious character level n-grams model	36
18	a snapshot of a keyword transformation n-grams model	37
19	A snapshot of the malicious n-grams composite word type transformation model	38
20	Java code added for model computation	39

21	CPU utilization before multithreading implementation	40
22	CPU idle time before multithreading implementation	40
23	CPU utilization after multithreading implementation	41
24	CPU idle time after multithreading implementation	41

CHAPTER 1

Introduction

JavaScript and its frameworks have been a 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, thus providing extensive functionalities such as form validation, animation, asynchronous behavior, user activity tracking, interactivity, and more. JavaScript is now progressively being used in server side code and in mobile applications by using cross-platform development tools like Titanium and PhoneGap [?].

Since the release of JavaScript in 1995, there have been many browsers and client-side security issues which have gained widespread attention [?]. 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 enable malicious actors to deliver scripts over the web and run them on client computers. Malicious JavaScript has been listed in the Open Web Application Security Project (OWASP)'s 2013 Top 10 List of security issues [?]. Cross-site scripting has been listed as the 3rd 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. Altogether the problem arises is how to classify malicious scripts from the benign ones accurately, as the dynamic nature of JavaScript makes it difficult to detect the exploit code. Moreover, the attackers utilize sophisticated obfuscation techniques that hide the malicious code and make detection complicated.

Recent work involves using machine learning techniques in combination with de-obfuscation/emulation technology [?]. Machine learning is used for feature extraction to identify the nature the scripts. The malicious code keeps evolving, taking benefits of the dynamic feature of JavaScript and sophisticated browser functionality. However, 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 Our approach to the problem

Our approach is based on TARDIS [?]. 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 TARDIS [?]. TARDIS uses machine learning techniques and robust features. These robust features 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 code generation algorithm, which further necessitates the use of additional resources from the attacker's part.

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 more likely to include more redundant parts as compared to a benign program. A benign, but inadequately written JavaScript program may also include redundancy and inefficiency. However, an attacker's intention of bypassing the detection of the exploit and the use of automation to generate obfuscated script tend to include added redundancy and inefficiency as compared to a benign and inefficient 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 [?]. SLM uses the general patterns in the language used in both malicious and benign JavaScript to classify benign and malicious JavaScript [?].

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 interact 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 websites that do not verify and sanitize user input 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 [?].

2.1.1 Stored Cross Site Scripting

Stored cross-site scripting targets the websites that store the user input first in databases or the file system and later reflect the user input to the web pages. If the input has not been sanitized or encoded and the data contains an attack, it will inject the attack in the web pages. This type of attack affects multiple users of the website [?].

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

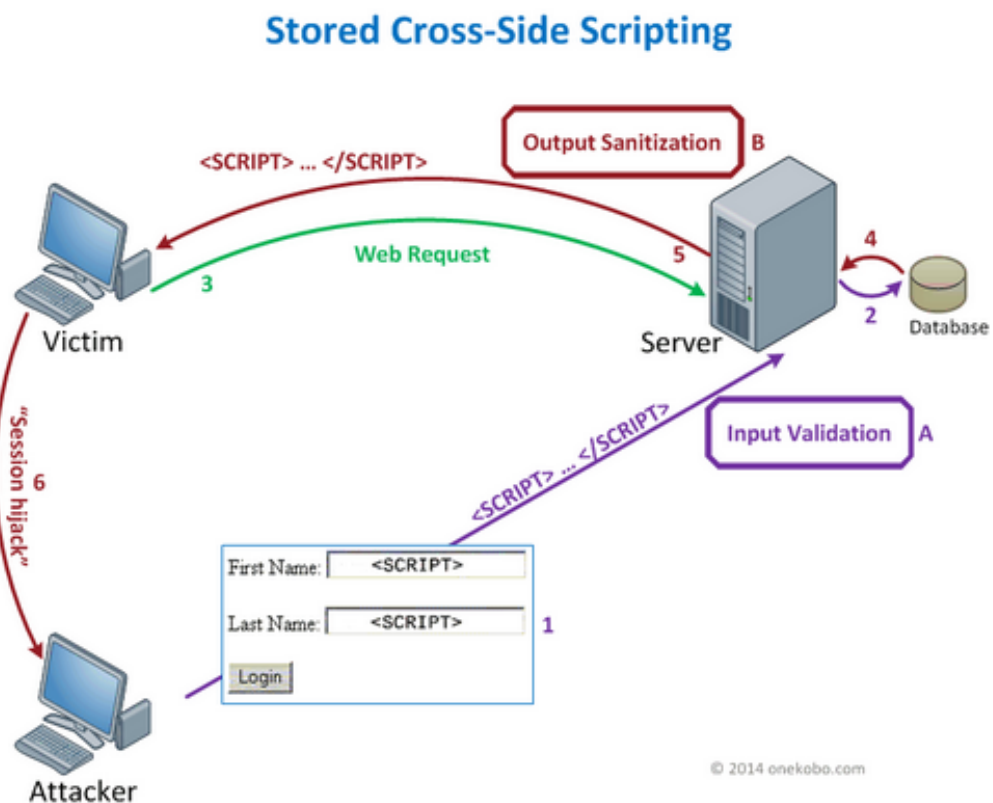


Figure 1: Stored cross-site scripting attack [?]

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 inject 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 [?].

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.

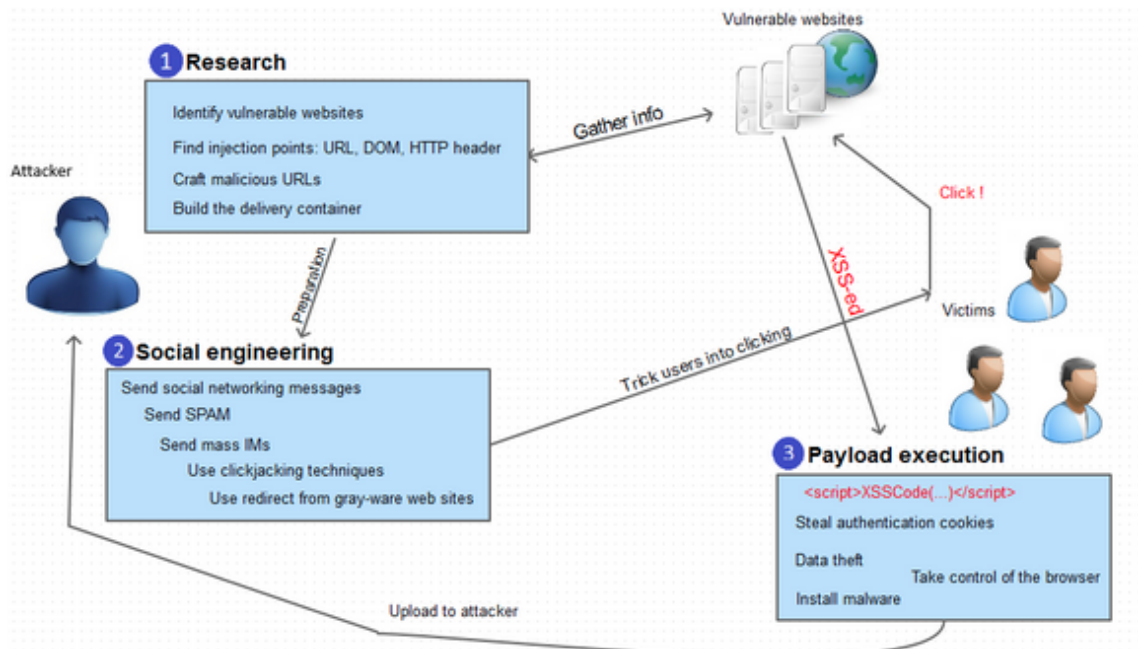


Figure 2: Reflected cross-site scripting attack [?]

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 [?].

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.substr(pos,document.URL.length));
</script>
...
</html>
```

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

<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 [?]

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 [?]

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 [?]

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 OWASP 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 [?]

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

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 taking the following measures to prevent or restrain a JavaScript attack: sandboxing and the same origin policy [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 by using the src attribute of a script tag in the top level document 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 [15]. Static analysis helps in understanding the composition of a program. The static analysis aims at examining the correctness and consistency of presentation and description of a software application, and serves as the first step in software quality control procedure [14]. 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 program. 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.

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.

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 approach is mostly 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 has 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 imple-

mented 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 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 of hierarchical features of the JavaScript abstract syntax tree 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 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 IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM [?]

IceShield 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 deobfuscation. 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 is capable of detecting the part of the internet page 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.

2.6.5 EarlyBird: Early Detection of Malicious Behavior in JavaScript Code [?]

EarlyBird 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 executed 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.

2.6.6 Wepawet [?]

Wepawet 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 [?]

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

Prophiler 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 methods usually in the order of tens of seconds per page. 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.

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;
      });

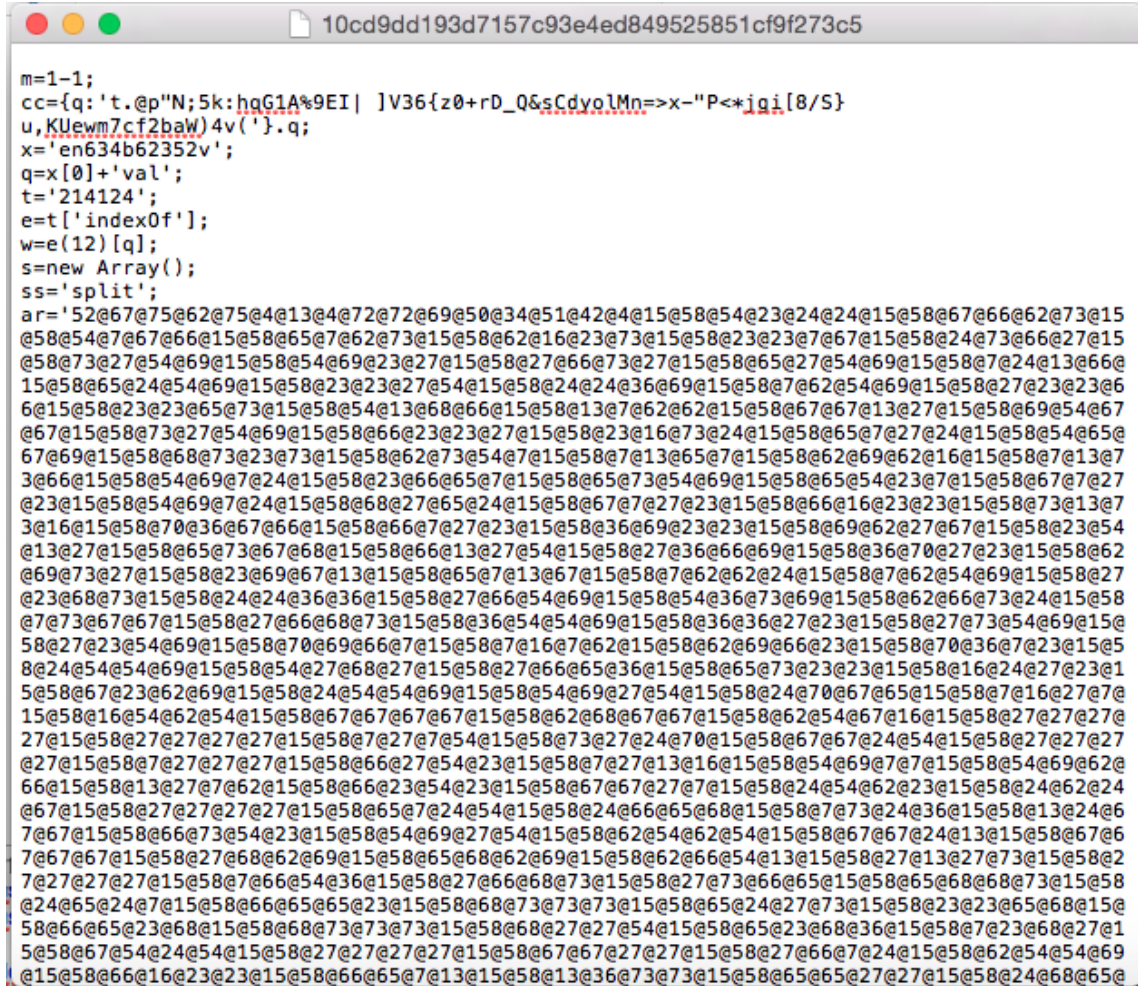
      promise.then(null, function (reason) {
        assert(reason instanceof TypeError);
        done();
      });
    });

    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 malicious script form test data set

A snapshot of a malicious JavaScript program



```
m=1-1;
cc={q:'t.@p"N;5k:hqG1A%9EI| ]V36{z0+rD_Q&sCdvo1Mn=>x-"P<*ig_i[8/5}
u,KUewm7cf2baW)4v(').q;
x='en634b62352v';
q=x[0]+'val';
t='214124';
e=t['index0f'];
w=e(12)[q];
s=new Array();
ss='split';
ar='52@67@75@62@75@4@13@4@72@72@69@50@34@51@42@4@15@58@54@23@24@24@15@58@67@66@62@73@15
@58@54@7@67@66@15@58@65@7@62@73@15@58@62@16@23@73@15@58@23@23@7@67@15@58@24@73@66@27@15
@58@73@27@54@69@15@58@54@69@23@27@15@58@27@66@73@27@15@58@65@27@54@69@15@58@7@24@13@66@
15@58@65@24@54@69@15@58@23@23@27@54@15@58@24@24@36@69@15@58@7@62@54@69@15@58@27@23@23@6
6@15@58@23@23@65@73@15@58@54@13@68@66@15@58@13@7@62@62@15@58@67@67@13@27@15@58@69@54@67
@67@15@58@73@27@54@69@15@58@66@23@23@27@15@58@23@16@73@24@15@58@65@7@27@24@15@58@54@65@
67@69@15@58@68@73@23@73@15@58@62@73@54@7@15@58@7@13@65@7@15@58@62@69@62@16@15@58@7@13@7
3@66@15@58@54@69@7@24@15@58@23@66@65@7@15@58@65@73@54@69@15@58@65@54@23@7@15@58@67@7@27
@23@15@58@54@69@7@24@15@58@68@27@65@24@15@58@67@7@27@23@15@58@66@16@23@23@15@58@73@13@7
3@16@15@58@70@36@67@66@15@58@66@7@27@23@15@58@36@69@23@23@15@58@69@62@27@67@15@58@23@54
@13@27@15@58@65@73@67@68@15@58@66@13@27@54@15@58@27@36@66@69@15@58@36@70@27@23@15@58@62
@69@73@27@15@58@23@69@67@13@15@58@65@7@13@67@15@58@7@62@62@24@15@58@7@62@54@69@15@58@27
@23@68@73@15@58@24@24@36@36@15@58@27@66@54@69@15@58@54@36@73@69@15@58@62@66@73@24@15@58
@7@73@67@67@15@58@27@66@68@73@15@58@36@54@54@69@15@58@36@36@27@23@15@58@27@73@54@69@15@
58@27@23@54@69@15@58@70@69@66@7@15@58@7@16@7@62@15@58@62@69@66@23@15@58@70@36@7@23@15@5
8@24@54@54@69@15@58@54@27@68@27@15@58@27@66@65@36@15@58@65@73@23@23@15@58@16@24@27@23@1
5@58@67@23@62@69@15@58@24@54@54@69@15@58@54@69@27@54@15@58@24@70@67@65@15@58@7@16@27@7@
15@58@16@54@62@54@15@58@67@67@67@67@15@58@62@68@67@67@15@58@62@54@67@16@15@58@27@27@27@
27@15@58@27@27@27@15@58@7@27@7@54@15@58@73@27@24@70@15@58@67@67@24@54@15@58@27@27@27
@27@15@58@7@27@27@27@15@58@66@27@54@23@15@58@7@27@13@16@15@58@54@69@7@7@15@58@54@69@62@
66@15@58@13@27@7@62@15@58@66@23@54@23@15@58@67@67@27@7@15@58@24@54@62@23@15@58@24@62@24
@67@15@58@27@27@27@27@15@58@65@7@24@54@15@58@24@66@65@68@15@58@7@73@24@36@15@58@13@24@6
7@67@15@58@66@73@54@23@15@58@54@69@27@54@15@58@62@54@62@54@15@58@67@67@24@13@15@58@67@6
7@67@67@15@58@27@68@62@69@15@58@65@68@62@69@15@58@62@66@54@13@15@58@27@13@27@73@15@58@2
7@27@27@27@15@58@7@66@54@36@15@58@27@66@68@73@15@58@27@73@66@65@15@58@65@68@68@73@15@58
@24@65@24@7@15@58@66@65@65@23@15@58@68@73@73@73@15@58@65@24@27@73@15@58@23@23@65@68@15@
58@66@65@23@68@15@58@68@73@73@73@15@58@68@27@27@54@15@58@65@23@68@36@15@58@7@23@68@27@1
5@58@67@54@24@54@15@58@27@27@27@27@15@58@67@67@27@27@15@58@27@66@7@24@15@58@62@54@54@69
@15@58@66@16@23@23@15@58@66@65@7@13@15@58@13@36@73@73@15@58@65@65@27@27@15@58@24@68@65@
```

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), (0xe2+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) [?] 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 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 represents the syntactic 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.

2.7.2 Statistical Language Modeling (SLM) [?]

Statistical language modeling makes use of a statistical language model. A statistical Language model is defined as a probability distribution of a string (s) in a sentence. 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 [20].

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 estimate the probability that a document belongs to a particular class (benign or malicious).

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 1, 2, 3, and 4 to tune the accuracy. N-grams of size 1 considers

each character as a feature while n-grams of size 2 joins together two consecutive characters. Similarly, a model based on n-grams of size 3 and four can be computed. N-grams model of size 1 tends to lose the surrounding context while n-grams model of a large size can provide too many surrounding contexts but less meaningful matches. Mostly n-grams of size 2 or 3 provide meaning full match with adequate surrounding contexts. TARDIS built its training model for n-grams of size 1 to n-grams of size 4 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 1, 2, 3, and 4.

2.7.5 Character level n-grams

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'], ['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 i = 1;
```

Keyword transformation n-grams of size 1 will look like

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

Keyword transformation n-grams of size 3 will look like

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

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 clas-

sifying a benign script if it is not obfuscated [?]. However, it does not prove very beneficial in identifying malicious, obfuscated scripts [?].

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 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 1 of 'var i = 1;'

['var', 'SPACE', 'i', 'SPACE', 'PUNCTUATION', 'SPACE', 'DIGIT', 'PUNCTUATION']

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

['var', 'SPACE', 'i'], ['SPACE', 'i', 'SPACE'], ['i', '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 [?].

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 [?], 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 [?].

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 [?].

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 [?]

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.

```
Anumehas-MacBook-Pro:my-addon anumehashah$ tree
.
├── Eclipse\ Installer.app
├── README.md
├── data
│   ├── CharacterKeywordTransform.js
│   ├── CompositeWordTransform.js
│   ├── SLM_Script.js
│   ├── image
│   │   ├── icon-16.png
│   │   ├── icon-32.png
│   │   ├── javascript.png
│   │   └── malware-icon.png
│   └── models
│       ├── benign
│       │   ├── CompositeBenignCountsA.js
│       │   ├── CompositeBenignCountsAB.js
│       │   ├── CompositeBenignTotal.js
│       │   ├── InputBenignCountsA.js
│       │   ├── InputBenignCountsAB.js
│       │   ├── InputBenignTotal.js
│       │   ├── KeywordBenignCountsA.js
│       │   ├── KeywordBenignCountsAB.js
│       │   └── KeywordBenignTotal.js
│       └── malicious
│           ├── CompositeMaliciousCountsA.js
│           ├── CompositeMaliciousCountsAB.js
│           ├── CompositeMaliciousTotal.js
│           ├── InputMaliciousCountsA.js
│           ├── InputMaliciousCountsAB.js
│           ├── InputMaliciousTotal.js
│           ├── KeywordMaliciousCountsA.js
│           ├── KeywordMaliciousCountsAB.js
│           └── KeywordMaliciousTotal.js
├── index.js
├── package.json
├── test
│   └── test-index.js
└── 6 directories, 30 files
Anumehas-MacBook-Pro:my-addon anumehashah$
```

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 3 and 4, keyword transformation n-grams of size 3 and 4, and composite word type transformation n-grams of size 3 and 4. 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`.

```
self.port.emit("script-response", "BenignAverageScore : " + benignAverageScore/scripts.length);  
self.port.emit("script-response", "maliciousAverageScore : " + maliciousAverageScore/scripts.length);
```

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

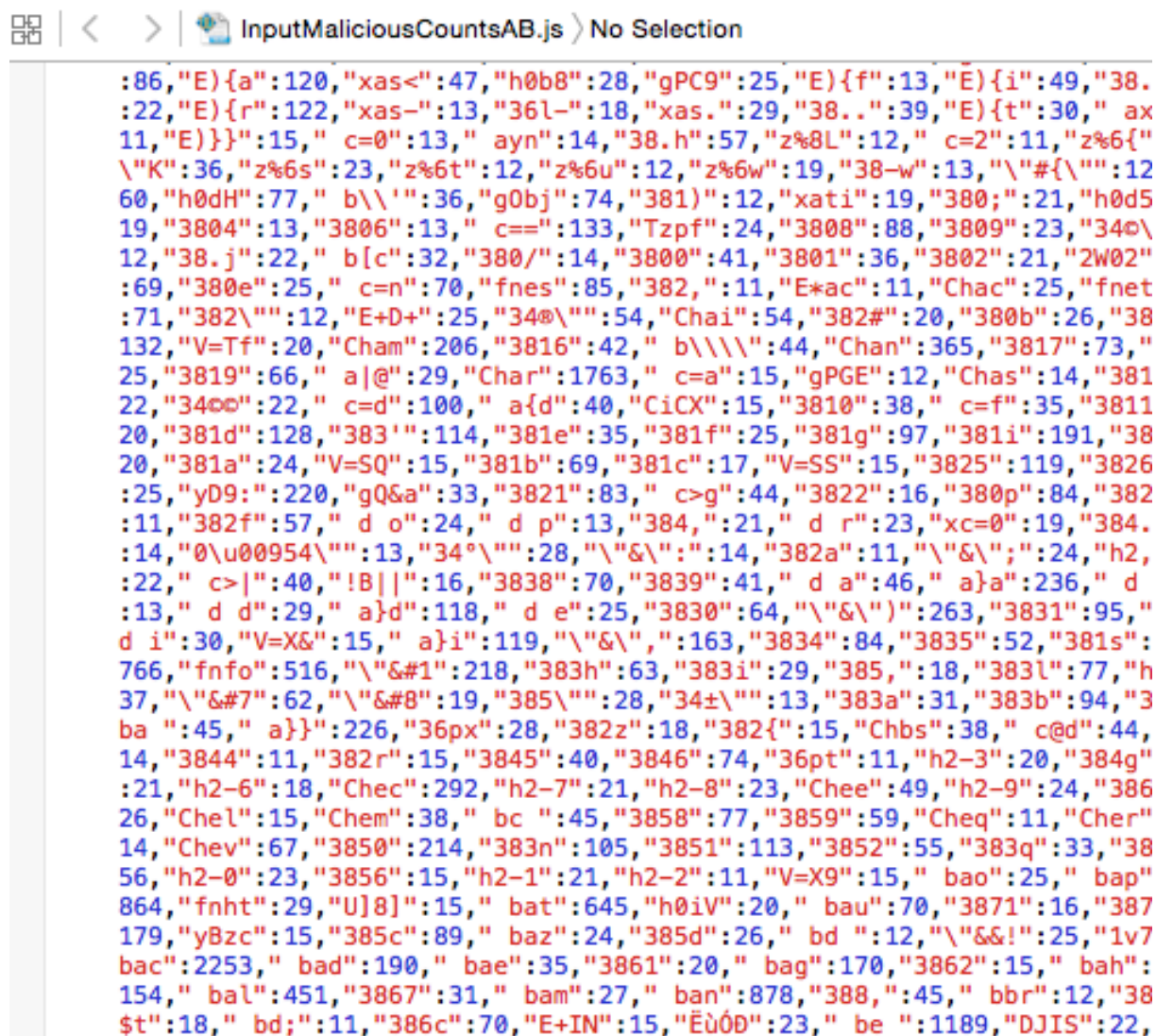
This section will present the detail discussion of the pre-compiled models we are utilizing for the add-on.

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.



```
:86,"E){a":120,"xas<":47,"h0b8":28,"gPC9":25,"E){f":13,"E){i":49,"38.  
:22,"E){r":122,"xas-":13,"36l-":18,"xas.":29,"38..":39,"E){t":30," ax  
11,"E)}}":15," c=0":13," ayn":14,"38.h":57,"z%8L":12," c=2":11,"z%6{  
\"K\":36,\"z%6s\":23,\"z%6t\":12,\"z%6u\":12,\"z%6w\":19,\"38-w\":13,\"\"#{\\\"\":12  
60,\"h0dH\":77,\" b\\\"\":36,\"g0bj\":74,\"381)\":12,\"xati\":19,\"380;\":21,\"h0d5  
19,\"3804\":13,\"3806\":13,\" c==\":133,\"Tzpf\":24,\"3808\":88,\"3809\":23,\"34c\\  
12,\"38.j\":22,\" b[c\":32,\"380/\":14,\"3800\":41,\"3801\":36,\"3802\":21,\"2W02\"  
:69,\"380e\":25,\" c=n\":70,\"fnes\":85,\"382,\":11,\"E*ac\":11,\"Chac\":25,\"fnet  
:71,\"382\\\"\":12,\"E+D+\":25,\"34@\\\"\":54,\"Chai\":54,\"382#\":20,\"380b\":26,\"38  
132,\"V=Tf\":20,\"Cham\":206,\"3816\":42,\" b\\\"\\\"\":44,\"Chan\":365,\"3817\":73,\"  
25,\"3819\":66,\" a|@\":29,\"Char\":1763,\" c=a\":15,\"gPGE\":12,\"Chas\":14,\"381  
22,\"34@\":22,\" c=d\":100,\" a{d\":40,\"CiCX\":15,\"3810\":38,\" c=f\":35,\"3811  
20,\"381d\":128,\"383'\":114,\"381e\":35,\"381f\":25,\"381g\":97,\"381i\":191,\"38  
20,\"381a\":24,\"V=SQ\":15,\"381b\":69,\"381c\":17,\"V=SS\":15,\"3825\":119,\"3826  
:25,\"yD9\":220,\"gQ&a\":33,\"3821\":83,\" c>g\":44,\"3822\":16,\"380p\":84,\"382  
:11,\"382f\":57,\" d o\":24,\" d p\":13,\"384,\":21,\" d r\":23,\"xc=0\":19,\"384.  
:14,\"0\\u00954\\\"\":13,\"34^\\\"\":28,\"\\\"&\\\"\":14,\"382a\":11,\"\\\"&\\\"\":24,\"h2,  
:22,\" c>|\":40,\"!B|\":16,\"3838\":70,\"3839\":41,\" d a\":46,\" a}a\":236,\" d  
:13,\" d d\":29,\" a}d\":118,\" d e\":25,\"3830\":64,\"\\\"&\\\"\":263,\"3831\":95,\"  
d i\":30,\"V=X&\":15,\" a}i\":119,\"\\\"&\\\"\":163,\"3834\":84,\"3835\":52,\"381s\":  
766,\"fnfo\":516,\"\\\"&#1\":218,\"383h\":63,\"383i\":29,\"385,\":18,\"383l\":77,\"h  
37,\"\\\"&#7\":62,\"\\\"&#8\":19,\"385\\\"\":28,\"34±\\\"\":13,\"383a\":31,\"383b\":94,\"3  
ba \":45,\" a}}\":226,\"36px\":28,\"382z\":18,\"382{\":15,\"Chbs\":38,\" c@d\":44,  
14,\"3844\":11,\"382r\":15,\"3845\":40,\"3846\":74,\"36pt\":11,\"h2-3\":20,\"384g\"  
:21,\"h2-6\":18,\"Chec\":292,\"h2-7\":21,\"h2-8\":23,\"Chee\":49,\"h2-9\":24,\"386  
26,\"Chel\":15,\"Chem\":38,\" bc \":45,\"3858\":77,\"3859\":59,\"Cheq\":11,\"Cher\"  
14,\"Chev\":67,\"3850\":214,\"383n\":105,\"3851\":113,\"3852\":55,\"383q\":33,\"38  
56,\"h2-0\":23,\"3856\":15,\"h2-1\":21,\"h2-2\":11,\"V=X9\":15,\" bao\":25,\" bap\"  
864,\"fnht\":29,\"U]8\":15,\" bat\":645,\"h0iV\":20,\" bau\":70,\"3871\":16,\"387  
179,\"yBzc\":15,\"385c\":89,\" baz\":24,\"385d\":26,\" bd \":12,\"\\\"&&!\":25,\"1v7  
bac\":2253,\" bad\":190,\" bae\":35,\"3861\":20,\" bag\":170,\"3862\":15,\" bah\":  
154,\" bal\":451,\"3867\":31,\" bam\":27,\" ban\":878,\"388,\":45,\" bbr\":12,\"38  
$t\":18,\" bd;\":11,\"386c\":70,\"E+IN\":15,\"ÈùÓÐ\":23,\" be \":1189,\"DJIS\":22,
```

Figure 17: A snapshot of a pre-compiled malicious character level n-grams model of size 4. 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.

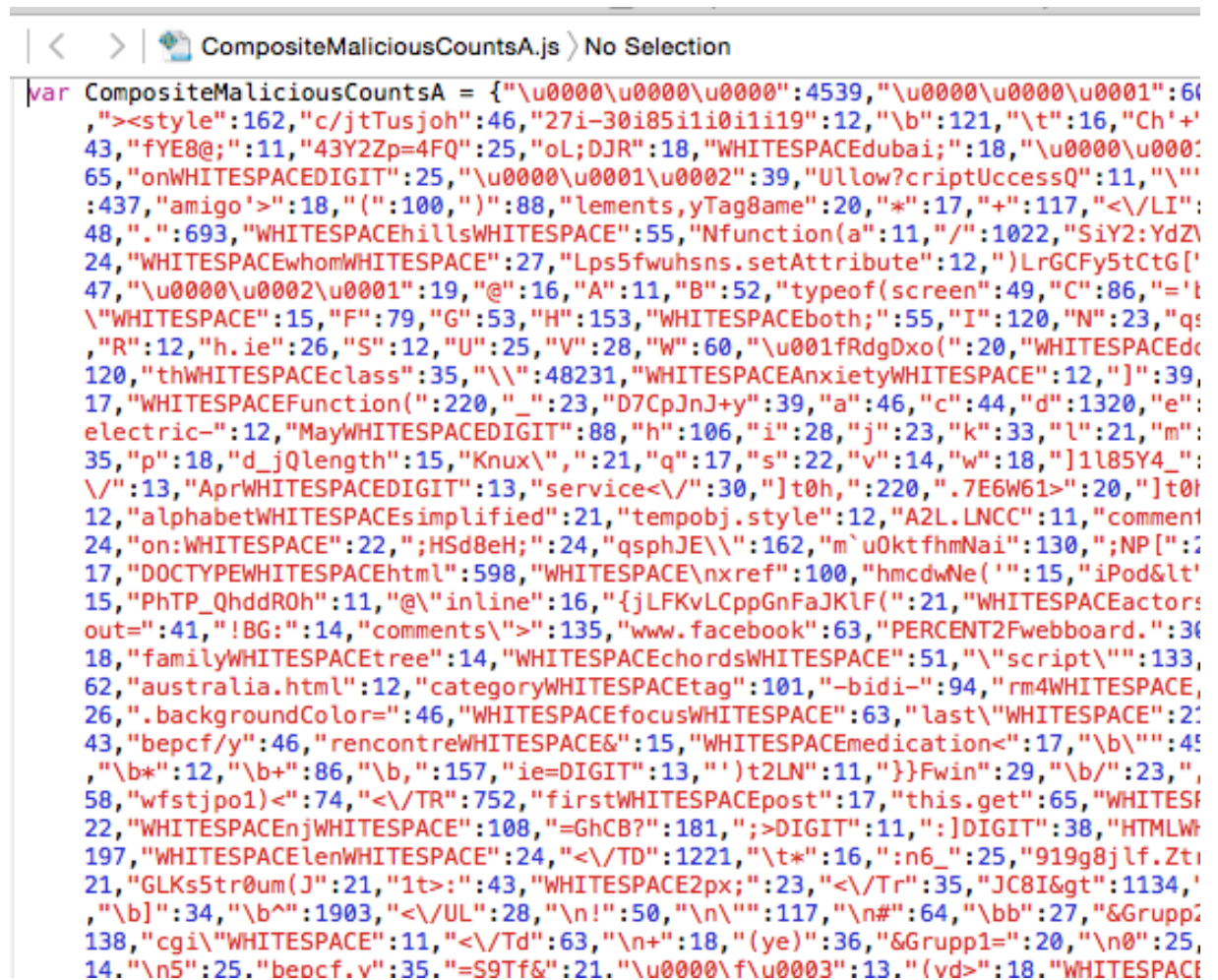


```
12, "fn[" :26, "38%2" :13, "h/z-" :12, "xc.5" :43, "fo:n" :15, "length<=0" :18, "\"";
16, "\"%26" :16, "38\""}" :17, "fo;\\" :11, "\"#pa" :24, "\"%2b" :16, "\"%2f" :16, "3
12, "\"%3c" :97, "\"%3f" :16, "length==" :467, "38'+":15, "length==0" :73, "\"#p
27, "length==3" :52, "fo;}" :32, "xc0)" :12, "xali" :20, "38'":58, "1];for" :37, "
, "\"#qu" :20, "fo=\\" :11, ".mindr" :14, "u:function(" :46, "\"#si" :14, "dfixed3
50, "\"#t=" :13, "h1>c" :12, "\"#sa" :16, "\"#te" :12, "fo=f" :14, "fo=g" :16, "fo=h'
11, "\"#ss" :14, "\"#st" :22, "xamp" :440, "h1><" :22, "fn]=" :11, "fn_g" :44, "38,\'
17, "fn_r" :16, "xc3\\" :49, "fn_s" :58, "this.ifr" :13, "38){":19, "fn_v" :23, "fn
133, "38,9" :67, "38,0" :289, "name.substrin" :36, "38,1" :515, "38,2" :244, "38,3'
177, "fn_" :12, "38,6" :187, "!--this" :29, "length;tc" :24, "fo?c" :11, "38,-":9
708, "38-1" :26, "xarr" :75, "length;u+" :17, "38+t" :22, "xat1" :26, "length;se":
16, ".min_v" :11, "length;rs" :12, "38.1" :13, "38.2" :31, "z%3q" :19, "38.5" :11, "
13, "380'" :11, "380\\" :13, "xatt" :127, "length;s+" :32, "fnca" :15, "\"#ya" :19, '
b" :11, "=\"constructor\\" :14, "?â?c" :14, "1to\\" :15, "3803" :39, "3804" :36, "3
37, "3809" :38, "length;r+" :76, "380:" :23, ".minim" :38, "380," :337, "38.j" :12, '
41, "ht(function" :43, "fnes" :23, "382," :293, "fnet" :450, "380]" :49, "fnex" :15
, "3817" :116, "3818" :33, "3819" :42, "381:" :23, "381;" :23, "381," :306, "3810" :3
32, "\u0000\u0000=\u0000" :785, "xavh" :20, "383," :297, "length;p+" :27, "381]"
44, "3826" :46, "3827" :34, "3828" :38, "3829" :59, "382:" :23, "xaw." :213, "length
39, "fndi" :16, "3823" :51, "3824" :36, "384'" :11, "384," :293, "382]" :49, "length
12, "typeofn==" :34, "1,dom" :11, "3836" :117, "fng_" :11, "3837" :77, "3838" :1715
16, "3830" :27, "length=0;" :115, "\"&\\" :120, "3831" :43, "36q1" :16, "length;n
21, "\"&\\" :61, "3833" :179, "36q3" :26, "length;n>" :11, "\"&\\" :47, "3834" :7
1251, "\"&#1" :11, "xaxi" :477, "385," :340, "385\\" :16, "length;n+" :67, "383a" :
15, "383c" :187, "383d" :727, "xaxe" :15, "383e" :245, "3847" :49, "3848" :46, "3849'
49, "3841" :46, "3842" :54, "3843" :52, "382q" :15, "3844" :38, "3845" :36, "3846" :3
324, "386." :14, "3860" :62, "de;if" :130, "!document.a" :38, "8°i±" :20, "length;r
23, "384]" :61, "3850" :47, "\"":this." :39, "3851" :27, "3852" :38, "3853" :45, "385
46, "3857" :32, "!document.i" :18, "387," :339, "3870" :51, "3871" :32, "387\\" :12
21, "3869" :57, "386:" :25, "385]" :61, "3861" :40, "3862" :45, "3863" :53, "3864" :4
```

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 three and four. As discussed in the section 2.7.4, composite word type transformation reduces entropy in an obfuscated malicious program.



```
var CompositeMaliciousCountsA = {"\u0000\u0000\u0000":4539,"\u0000\u0000\u0001":60, "><style":162,"c/jtTusjoh":46,"27i-30i85i1i0i1i19":12,"\b":121,"\t":16,"Ch'+43,"fYE8@":11,"43Y2Zp=4FQ":25,"oL;DJR":18,"WHITESPACEdubai":18,"\u0000\u000065,"onWHITESPACEDIGIT":25,"\u0000\u0001\u0002":39,"Ullow?criptUccessQ":11,"\":437,"amigo'>":18,"(":100,")":88,"lements,yTag8ame":20,"*":17,"+":117,"<\LI":48,"":693,"WHITESPACEhillsWHITESPACE":55,"Nfunction(a":11,"/":1022,"SiY2:YdZl24,"WHITESPACEwhomWHITESPACE":27,"Lps5fwuhsns.setAttribute":12,")LrGCFy5tCtG['47,"\u0000\u0002\u0001":19,"@":16,"A":11,"B":52,"typeof(screen":49,"C":86,"='t\"WHITESPACE":15,"F":79,"G":53,"H":153,"WHITESPACEboth":55,"I":120,"N":23,"qs,\"R":12,"h.ie":26,"S":12,"U":25,"V":28,"W":60,"\u001fRdgDxo(":20,"WHITESPACEdc120,"thWHITESPACEclass":35,"\\":48231,"WHITESPACEAnxietyWHITESPACE":12,"]":39,17,"WHITESPACEFunction(":220,"_":23,"D7CpJnJ+y":39,"a":46,"c":44,"d":1320,"e":electric-":12,"MayWHITESPACEDIGIT":88,"h":106,"i":28,"j":23,"k":33,"l":21,"m":35,"p":18,"d_jQlength":15,"Knux\\":21,"q":17,"s":22,"v":14,"w":18,"]1l85Y4_":\\":13,"AprWHITESPACEDIGIT":13,"service</":30,"]t0h,"":220,".7E6W61>":20,"]t0f12,"alphabetWHITESPACESimplified":21,"tempobj.style":12,"A2L.LNCC":11,"comment24,"on:WHITESPACE":22,";Hsd8eH":24,"qsphJE\\":162,"m`u0ktfhmNai":130,";NP[":217,"DOCTYPEWHITESPACEhtml":598,"WHITESPACE\\nxref":100,"hmc dwNe('":15,"iPod&lt'15,"PhTP_QhddR0h":11,"@\\inline":16,"{jLFKvLCppGnFaJKlF(":21,"WHITESPACEactor:out=":41,"!BG=":14,"comments\\>":135,"www.facebook":63,"PERCENT2Fwebboard."":3618,"familyWHITESPACETree":14,"WHITESPACEchordsWHITESPACE":51,"\\script\\":133,62,"australia.html":12,"categoryWHITESPACETag":101,"-bidi-":94,"rm4WHITESPACE,26,".backgroundColor=":46,"WHITESPACEfocusWHITESPACE":63,"last\\WHITESPACE":2:43,"becpf/y":46,"rencontrewHITESPACE&":15,"WHITESPACEmedication<":17,"\\b\\":45,"\\b*":12,"\\b+":86,"\\b,"":157,"ie=DIGIT":13,"')t2LN":11,"}}Fwin":29,"\\b/":23,"58,"wfstjpol)<":74,"<\\TR":752,"firstWHITESPACEpost":17,"this.get":65,"WHITESF22,"WHITESPACenjWHITESPACE":108,"=GhCB?":181,";>DIGIT":11,":]DIGIT":38,"HTMLWf197,"WHITESPACElenWHITESPACE":24,"<\\TD":1221,"\\t*":16,"n6_":25,"919g8jlf.Zt121,"GLKs5tr0um(J":21,"1t>":43,"WHITESPACE2px":23,"<\\Tr":35,"JC8I&gt":1134,',"\\b]:34,"\\b^":1903,"<\\UL":28,"\\n!":50,"\\n\\":117,"\\n#":64,"\\bb":27,"&Grupp2138,"cgi\\WHITESPACE":11,"<\\Td":63,"\\n+":18,"(ye)":36,"&Grupp1=":20,"\\n0":25,14,"\\n5":25,"becpf.v":35,"=S9Tf&":21,"\\u0000\\f\\u0003":13,"(vd>":18,"WHITESPACE
```

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 is light weight 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_maliciousCountsAB = new JSONObject();
for(Map.Entry<TermSequence, Integer> entry : maliciousModel.countsAB.entrySet()){
    String key = entry.getKey().toString();
    key = key.substring(1, key.length()-1).replace(" ", "");

    int value = entry.getValue();

    if(value > 10000){

        System.out.printf("Key : %s and Value: %s %n", key, entry.getValue());
    }
    try {
        if(value > 10)
            obj_maliciousCountsAB.put(key, value);
    } catch (JSONException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

FileWriter file_MaliciousCountsAB;
try {
    file_MaliciousCountsAB = new FileWriter("inputMaliciousCountsAB.json");
    file_MaliciousCountsAB.write(obj_maliciousCountsAB.toString());
    file_MaliciousCountsAB.flush();
    file_MaliciousCountsAB.close();
} catch (IOException e2) {
    // TODO Auto-generated catch block
    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 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 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    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    0  1  2  0  0 100  0  0
1 0      0 23312496 3128 631696  0  0    0  0    0 1117 365  8  0 92  0  0
1 0      0 23312592 3128 631744  0  0    0  0    0 1088 314  8  0 92  0  0
1 0      0 23312592 3128 631744  0  0    0  0    1 1080 291  8  0 92  0  0
1 0      0 23312592 3128 631744  0  0    0  0    0 1078 289  8  0 92  0  0
1 0      0 23312592 3128 631744  0  0    0  0    0 1075 271  8  0 92  0  0
1 0      0 23312592 3128 631744  0  0    0  0    0 1085 301  8  0 92  0  0
1 0      0 23312592 3128 631744  0  0    0  0    0 1072 270  8  0 92  0  0
1 0      0 23312592 3128 631744  0  0    0  0    0 1073 284  8  0 92  0  0
1 0      0 23312716 3128 631744  0  0    0  0    0 1070 276  8  0 92  0  0
1 0      0 23312716 3128 631744  0  0    0  0    0 1116 357  8  0 92  0  0
2 0      0 23312644 3128 631788  0  0    0  0    0 1083 318  8  0 92  0  0
1 0      0 23312676 3128 631800  0  0    0  0    0 1088 310  8  0 92  0  0
1 0      0 23312676 3128 631800  0  0    0  0    1 1071 285  8  0 92  0  0
1 0      0 23312676 3128 631800  0  0    0  0    0 1096 306  8  0 92  0  0
1 0      0 23312676 3128 631800  0  0    0  0    0 1072 303  8  0 92  0  0
1 0      0 23312676 3128 631800  0  0    0  0    0 1113 318  9  0 92  0  0
1 0      0 23312644 3128 631812  0  0    0  0    0 1071 283  8  0 92  0  0
1 0      0 23312644 3128 631812  0  0    0  0    0 1091 301  8  0 92  0  0
0 1      0 23312644 3128 631816  0  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 0 1 3	0 97 0 0
4 0	0 20637880 3128 3290476	0 0	0 0	0 4245 28358 22	2 76 0 0
5 0	0 20637880 3128 3290476	0 0	0 0	0 4600 29676 22	2 76 0 0
4 0	0 20637880 3128 3290476	0 0	0 0	0 4509 25998 22	2 77 0 0
3 0	0 20611008 3128 3290476	0 0	0 0	0 4899 22013 23	4 73 0 0
2 0	0 20611008 3128 3290476	0 0	0 0	0 3641 28899 18	2 80 0 0
3 0	0 20611008 3128 3290476	0 0	0 0	0 3415 30799 16	2 82 0 0
3 0	0 20611008 3128 3290476	0 0	0 0	0 3663 31191 17	2 81 0 0

Figure 24: CPU idle time after multithreading implementation = 76

3.10 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 3 model and n-grams of size 4 JSON model. The model then computes the overall probability of the script for both the benign and malicious models using the formula

$$Probability = probability + \text{math.log}(pAB/pA)$$

$$pAB = (\text{frequency of n-grams of size 4}) / (\text{total no of words})$$

$$pA = (\text{frequency of the n-grams of size 3}) / (\text{total no of words})$$

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 our 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 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 megabytes 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.

APPENDIX A

Zorak Likes Beans

A.1 Oh Yes He Does

Appendices can have sections and subsections and so on.

A.2 Really

Sections, subsections, or whatever should come in pairs.

APPENDIX B

Everybody Wants to Be Space Ghost

Space Ghost: Everybody wants to be Space Ghost

Everybody near and far

Hey, ma, look at me, I'm on TV

Everybody wants to be a star

I'm Space Ghost, Mr. Space Ghost

I've got big muscles, And I can dance

When Zorak tries to bug me, I zap him with my power bands

Zorak: Uhhhh... I don't think that will be necessary

Space Ghost: I think I'll zap him with my power bands

On Saturn and Jupiter and Neptune, too

I've been hearing it from coast to coast

I'd give anything, If for just one day

I could be a super hero like Space Ghost

Zorak: Ugghh... Why would anyone want to be Space Ghost?

Space Ghost: Everybody wants to be just like me!