



Faculty of Computer Science and Information Technology

MalSketch – A Machine Learning-Based Malware Behaviour Analysis Framework

Navein Chanderan

**Master of Science
2019**

MalSketch – A Fast and Efficient Machine Learning-Based Malware Behaviour Analysis

Navein Chandaran

A thesis submitted

In fulfilment of the requirements for the degree of Master of Science

(Computer Science)

Faculty of Computer Science and Information Technology
UNIVERSITI MALAYSIA SARAWAK

2019

DECLARATION

I hereby declare that the thesis is based on my original work except for quotations and citation, which have been duly acknowledged. The thesis has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

Name: Navein Chanderan

Matric No.: 15020358

Faculty of Computer Science and Information Technology

Universiti Malaysia Sarawak

DEDICATION

I would like to give my praise to God Almighty for the love, grace, and mercy He bestowed upon me during the past two years. I thank Him for giving me a very loving and supportive family who has encouraged and supported me all the way. I thank Him for giving me an excellent supervisor, Associate Professor Dr. Johari Abdullah, who has guided me and helped me a lot in my journey, and for being a good mentor to me. I would also like to thank all the caring and supportive friends out there who has made my life more cheerful and colourful during the course of my journey here.

ACKNOWLEDGEMENT

I would like to thank UNIMAS for funding this research under Research Grant: F08/SpFRC/1432/16/6. I would also like to thank VirusTotal for providing me with a special academic access to their service which allowed me to complete my research by using their services to download malware metadata and labels for my research dataset.

ABSTRACT

Malware samples has increased exponentially over the years, and there is a need to improve the efficiency of analysing large number of malware samples. Additionally, the diversity of malware types and the methods it employs to defeat analysis techniques has also increased steadily. Static analysis methods of malware are just not enough to combat modern malware attack as it has inherent limitation in that it is easily defeated by obfuscation and polymorphism. On the other hand, dynamic analysis methods of malware behaviour do not suffer from such limitations due to the fact that the samples are executed, therefore revealing its true behaviours. To address this problem, a framework for the automatic analysis of malware behaviour is proposed. The framework analyses malware behaviour, then convert the behaviour reports into a metalanguage format suitable for machine learning. To speed up computation, Minhash is used to represent samples, and Locality Sensitive Hashing is applied for nearest neighbour search in sublinear time. Disjoint-set Forest clustering algorithm is then applied to the results to cluster malware into family clusters. The framework achieves 97.4% true positive rate and 99.4% true negative rate, using a dataset of 65,000 from VirusShare. This shows that the framework works very well even for random dataset, and it is capable of daily malware samples clustering and to identify unknown malware.

Keywords: Malware behaviour, malware analysis, clustering, automated analysis

MalSketch - Analisis Kelakuan Malware Menggunakan Pembelajaran Mesin

ABSTRAK

Jumlah perisian komputer perusak (malware) bertambah secara eksponen tahun demi tahun, maka, adalah perlu untuk menambahbaik kecekapan menganalisis jumlah malware yang banyak. Jenis teknik yang digunakan oleh malware untuk megalahkan teknik analisis juga telah bertambah banyak. Teknik analisis statik adalah tidak berkesan untuk menangani malware yang kompleks dan mempunyai kapabiliti menyembunyi kod menggunakan teknik enkripsi dan polimorfisma. Manakala, teknik analisis dinamik tidak mempunyai kelemahan yang sama, kerana malware yang dieksekusi akan memberi gambaran yang sebenar. Untuk menyelesaikan masalah ini, rangka kerja untuk analisis kelakuan perisian perusak automatik dicadangkan. Rangka kerja ini akan menganalisis kelakuan malware, menukarkan format laporan analisis kepada format metabahasa yang sesuai untuk pembelajaran mesin. Untuk meningkatkan proses komputasi, Minhash digunakan untuk mewakili malware, dan 'Locality Sensitive Hashing' diaplikasikan untuk mempercepatkan pencarian kelakuan malware yang hampir sama dalam masa sublinear. Algoritma 'Disjoint-set Forest' kemudian diaplikasikan untuk mengkluster malware dalam kluster keluarga. Pendekatan ini mencapai kadar positif benar 97.4% dan kadar negatif benar 99.4%, apabila set data yang mempunyai 65,000 sampel malware dari VirusShare digunakan untuk analisis. Ini menunjukkan bahawa pendekatan ini amat berkesan walaupun data yang dianalisis adalah rawak, dan ianya mempunyai kemampuan untuk pengklusteran harian sampel malware, dan untuk mengenalpasti malware yang tidak diketahui.

Kata kunci: Kelakuan malware, analisis malware, pengklusteran, analisis automatik

TABLE OF CONTENTS

	Page
DECLARATION	i
DEDICATION	ii
ACKNOWLEDGEMENT	iii
ABSTRACT	iv
<i>ABSTRAK</i>	v
LIST OF TABLES	ix
LIST OF FIGURES	x
CHAPTER 1: INTRODUCTION	1
1.1 Overview	1
1.2 Problem Statement	4
1.3 Objective	5
1.4 Scope and Limitations	6
1.5 Significance and Contribution	6
1.6 Thesis Outline	7
CHAPTER 2: LITERATURE REVIEW	8
2.1 Overview	8
2.2 Categories of Malware	8
2.3 Purpose and Types of Malware Analysis	10
2.3.1 Static Analysis	11
2.3.2 Dynamic Analysis	12
2.3.3 Discussion on Malware Analysis Techniques	13

2.4	Related Works	15
2.4.1	Supervised Learning	15
2.4.2	Unsupervised Learning	17
2.4.3	Feature Selection, Extraction and Representation	19
2.4.4	Malware Labelling	23
2.5	Discussion	24
2.6	Summary	30
	CHAPTER 3: METHODOLOGY AND FRAMEWORK	32
3.1	Introduction	32
3.2	MalSketch Overview	32
3.3	Framework Components	33
3.3.1	Database	34
3.3.2	Sandbox	35
3.3.3	Malware Instruction Set	42
3.3.4	W-Shingles and Minhash	47
3.3.5	Locality Sensitive Hashing	49
3.3.6	Clustering	52
3.3.6.1	Disjoint-Set Forest	52
3.3.6.2	Single-linkage Hierarchical Clustering	55
3.3.6.3	DBSCAN	56
3.3.6.4	HDBSCAN	57
3.4	Summary	59
	CHAPTER 4: EXPERIMENTS, RESULTS AND ANALYSIS	60
4.1	Introduction	60

4.2	Experiment on Minhash Signature	61
4.3	Experiment on Locality Sensitive Hashing	64
4.4	Experiment on Clustering	66
4.5	Experiment of the MalSketch against Malheur	69
4.6	Summary	71
CHAPTER 5: CONCLUSION AND FUTURE WORKS		72
5.1	Summary	72
5.2	Future Works	73
5.3	Conclusion	74
REFERENCES		75
APPENDICES		84

LIST OF TABLES

	Page
Table 2.1 Malware Categories	9
Table 2.2 Comparison of related works	24
Table 3.1 MIST categories, encoding number, and unique operations for each category	44
Table 3.2 S-curve probability based on $b = 20$ and $r = 20$, $t = 0.9$, <i>and step value ≈ 0.86</i>	51
Table 4.1 Minhash evaluation parameters	61
Table 4.2 Experiment criteria for determining true positive true negative rate, and query time using Malheur dataset	64
Table 4.3 Results of LSH experiment on Malheur dataset	65
Table 4.4 Experiment criteria to measure nearest neighbour search speed	66
Table 4.5 Results of nearest neighbour search experiment	66
Table 4.6 Experiment criteria for cluster quality	67
Table 4.7 Results of cluster quality experiment	69
Table 4.8 Experiment to determine the true positive and true negative rates, and clustering time of Malheur and MalSketch	69
Table 4.9 Results of true positive and true negative rates, and clustering time of Malheur vs MalSketch	71

LIST OF FIGURES

	Page
Figure 1.1 Yearly total malware samples since 2010 until 2019 (AVTest, 2019)	2
Figure 3.1 MalSketch Overview	33
Figure 3.2 ERD of the framework database	34
Figure 3.3 Overview of single instance Cuckoo Sandbox	36
Figure 3.4 Overview architecture of distributed Cuckoo Sandbox analysis	37
Figure 3.5 Pseudocode for labelling samples	40
Figure 3.6 Snippet of the original behaviour report	41
Figure 3.7 MIST schematic format	43
Figure 3.8 Pseudocode for Murmurhash3 32bit	45
Figure 3.9 Snippet of MIST report format	46
Figure 3.10 LSH S-curve diagram, (Leskovec et al., 2014)	51
Figure 3.11 Pseudocode for Make Set	53
Figure 3.12 Pseudocode for Find with Path Splitting	54
Figure 3.13 Pseudocode for Union by Rank	54
Figure 3.14 Single-linkage Hierarchical clustering pseudocode	55
Figure 3.15 DBSCAN pseudocode	57
Figure 3.16 HDBSCAN pseudocode	58
Figure 4.1 Accuracy of Minhash signature based on number of permutations	62
Figure 4.2 Performance of Minhash in terms of running time relative to number of permutation functions	63

CHAPTER 1

INTRODUCTION

1.1 Overview

Computer systems and the Internet have become ubiquitous in this modern era. As the fourth industrial revolution is set in motion, it will become more intertwined with our daily life more than ever (Jazdi, 2014). As it becomes an increasingly important medium of information transmission, retention and connectivity, cyber security threats arise. These security threats are usually in the form of malware, also known as malicious software, which refers to various kinds of hostile, intrusive, destructive or even annoying software which performs unintended actions depending on its design and purpose. In the past, malware was created for fun and for showing off skills, but due to the lucrative economy of cybercrimes, it has turned into something to be used for very sinister purposes by various parties such as cybercriminal groups, nation-states, and even business rivals.

According to AV-Test, it registers over 350,000 samples daily (AV-Test GmbH, 2015). Most of the malware samples are variants of existing malware families. With the rise of malware creation tools in the black market, it is very easy for cybercriminals to create malware, even without knowledge on cyber security. Symantec reported that there are 431 million new variants of malware detected in year 2015, which shows a 36% rise over previous year (Symantec Corporation, 2016). Due to the lucrative cybercrime economy, malware threats will only keep growing. Figure 1.1 shows the exponentially rising trend of malware across the years.

Total malware

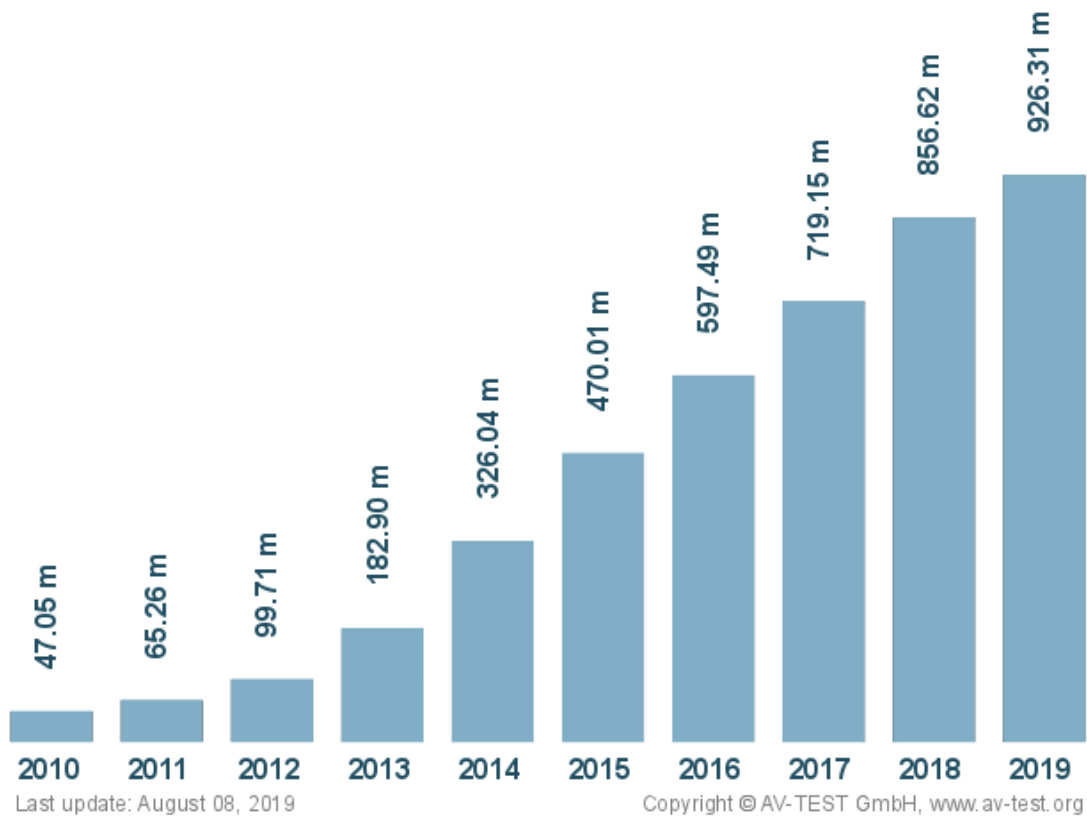


Figure 1.1: Yearly total malware samples since 2010 until 2019 (AVTest, 2019)

This has led to many negative consequences, such as massive monetary losses and downtime for critical services in recent years. Malware attacks range from compromising vulnerable computers, penetrating computer networks, stealing confidential information, sending out spam emails, blackmailing, to crippling critical infrastructures or services. In 2015, an Intel-security study result indicates that the annual cost of cybercrime to the global economy to be at an estimated US\$400 billion (McAfee, 2018), and by 2019, it is predicted that it will exponentially grow to US\$2.1 trillion (Juniper Research, 2018).

Due to the exponentially increasing and evolving dangers from malware threats, it has created a challenge for security vendors to deal with analysing the huge influx of malware efficiently. This in turn has created another challenge for the industry players to protect computer systems and network resources from the threats of malware more promptly. There are various kinds of analysis tools that security professionals can utilise to perform malware analysis, but generally, all the tools fall under one of the two main categories of analysis methods, which are content-based static analysis and behaviour based dynamic analysis (Zhu & Yin, 2012). Both approaches have the same goal of identifying how malware works, but the methods, tools, and skills needed for it differs.

Static analysis technique is the analysis of malware binary without executing it, by examining the structure of the malware codes. It has the advantage of providing good insights into the working of malwares based on the instructions in the reversed engineered source codes. Dynamic analysis technique is the analysis of malware behaviour by execution of the sample in a controlled environment, usually in virtual machines and sandboxes. Patterns of executed malware behaviour sequence can be observed, and the intent can be understood more easily as opposed to analysing source codes in static analysis. It is also more resilient to obfuscation and encryption techniques which could renders static analysis unable to analyse certain samples. However, these two basic techniques on its own are not sufficient for large scale analysis.

Therefore, the main objective of this thesis is to enable large scale analysis of malware samples. To achieve this goal, the objective is to study and evaluate a machine learning-based anti-malware framework to analyse large numbers of malware behaviour reports derived from dynamic analysis using sandbox. The framework could discriminate between

known and unknown malware samples and lead to the discovery of unknown samples. This can lead to time savings for analysis and reduce analyst workloads to only analyse unique unknown samples.

1.2 Problem Statement

Moser, Kruegel, and Kirda (2014) shows that static analysis has drawbacks due to the fundamental limits in what can be decided statically, due to code obfuscation and polymorphism. Modern malwares are usually packed and obfuscated in order to thwart analysis. There are generic unpacking and de-obfuscation tools available, but some malwares employ custom packers and de-obfuscators, which makes reversing the protection much harder and time consuming. Moser, Kruegel, and Kirda (2007) proved that analysing obfuscated malware binaries can yield inaccurate information when opaque constants are used, as the disassembled binaries cannot be analysed precisely. It is also confirmed by Sihwail, Omar and Zainol Ariffin (2018) that is static analysis can be defeated using obfuscation. The problems which affects static analysis will hinder the effectiveness of this approach (Egele, Scholte, Kirda, & Kruegel, 2012).

On the other hand, dynamic analysis has an advantage against malware samples, especially those which are obfuscated and complex in structure. It can reveal information which cannot be inferred by using static analysis methods as stated by Sihwail, Omar and Zainol Ariffin (2018). Runtime debugging method of dynamic analysis which debugs the binary codes is usually quite accurate as most of the codes can be de-obfuscated. However, it is not independent to unpacking, making it not reliable as unpackers cannot possibly unpack every kind of packers used to obfuscate the binaries. On the other hand, runtime behaviour

monitoring approach does not have such limitations. This would make it a better approach to use, as it is more independent to code obfuscation and independent of unpacking malware binaries (Rieck, Trinius, Willems, & Holz, 2011).

Based on the known strengths and weaknesses of each analysis method, dynamic analysis method which is based on runtime behaviour monitoring has advantages over static analysis and runtime debugging-based dynamic analysis. However, it still does not solve the problem of analysing samples at a big scale. As the number of malwares detected in the wild are increasing exponentially, this is a severe problem to be overcome. With the substantial number of malware samples amounting to around 370,000 samples daily, it is not possible for analysts to analyse the large number of samples as it is very time consuming to analyse malware samples based on regular static analysis and dynamic analysis. A more efficient method is needed to identify similar malware samples and map the family relationship and its unique indicators. This would allow analysts to prioritise and spend more productive time on analysing only unknown or important samples.

1.3 Objective

The main objective of this thesis is to study and design a lightweight automated machine learning-based malware analysis triaging framework that can analyse and identify malware, based on the behaviour generated by malware sandbox analysis.

To achieve the main objective, the following are several specific objectives that must be fulfilled, which are:

- a. To determine malware sample behaviour patterns through sandbox analysis.
- b. To study malware metalanguage format to be used to convert behaviour reports for efficient machine learning, and unsupervised machine learning algorithm to efficiently cluster behaviour patterns generated by the malware metalanguage framework.
- c. To evaluate the performance of the framework.

1.4 Scope and Limitations

For the scope of this research, it will focus only on Windows-based malware, researching on a suitable malware metalanguage representation and unsupervised machine learning algorithm.

As for the limitations of this research, malware without labels will not be labelled, and no label normalisation will be done for those samples. This research also does not focus benign software behaviour analysis and comparison.

1.5 Significance and Contribution

This research focuses on creating an automated lightweight analysis framework of malware samples based on its behaviour, using efficient machine learning approach. It will serve as a foundation for future automated large-scale malware analysis service platform, as planned by this research.

1.6 Thesis Outline

This thesis is structured into five chapters and is explained as follows;

- Chapter 1 describes the introduction to this research work.
- Chapter 2 is the literature review, it explains malware analysis, malware metadata format, nearest neighbour search and machine learning algorithms, and discussions on the related works in greater details.
- Chapter 3 presents the methodology and framework of this research.
- Chapter 4 shows the experiments conducted, the results of the experiments, and the analysis of the results.
- Chapter 5 contains the summary, conclusion, and future work of this research.

CHAPTER 2

LITERATURE REVIEW

2.1 Overview

In this chapter, reviews of literatures are done. The chapter will contain in-depth analysis and discussion of the categories of malware, the methods used in analysis, and the limitations of the basic methods. This chapter will also discuss each part in detail which constitutes the basic blocks of the machine learning framework. Related works are also discussed in detail, which will lead to a discussion of what should be improved, such as methods that solves the problem of efficiency and scale, and what should be done to solve the problem. The summary then details what will be done for this research, which will lead to the solutions proposed in Chapter 3.

2.2 Categories of Malware

Malware has a long history in the computing world, even before the widespread availability of Internet connectivity. In the early days, malware is just a simple piece of software, usually written with the objective of showing off skills or to have fun. Since the late 1990s however, it has changed. Malware has since grown more stealth and sophisticated, and used as a tool for various kinds of cybercrimes.

There are many variations of malware whether known, or in the wild. Malware will continue to evolve, and some modern malware do not fit a single classification family due to its

ambiguous nature. However, malware can still be categorised based on its action. The table below explains the most common malware categories:

Table 2.1: Malware Categories

Malware type	Definition	Author & Year
Virus	A type of malicious software which infects the host by replicating itself recursively onto host files, software, or system area by modifying them to include a possibly evolved version of itself	Horton, Jeffrey, & Seberry, 1997
Trojan	A type of malicious software which portrays itself to do something but in actuality doing something else. Trojan can compromise computer systems by opening backdoors for hackers or other malware to spread into networks or steal personal/confidential information. It is usually encoded in a hidden payload, where the host containing it may be a cleanware, or disguising as a legitimate software. It cannot replicate itself like a computer virus	Symantec.com, n.d.
Worm	A type of malicious software which can self-replicate and propagate via computer networks, without human intervention. An important distinction between virus and worm is that virus requires a host to attach itself to in order to replicate, while worm is a stand-alone program	Kaspersky, 2018
Rootkit	A malicious program designed to infect computer systems stealthily and with persistence. It is usually designed to reside deep in operating systems kernel or firmware with administrative rights, making it very hard to detect and also remove. Rootkits may contain tools like keylogger, password stealer, banking or credit card stealer, or even backdoor to allow attackers to infiltrate into a computer system	Kaspersky, 2018
Spyware	A type of malicious software which is designed to spy on users by scanning or monitoring and steal information and conveying the stolen information to third parties, without the consent of the user. The type of information can be login details, passwords, personal information, banking details, specific file, or even information about the user's computer such as applications installed on the computer, computing habits	Kaspersky, 2018

Table 2.1 continued

Keylogger	A type of software which logs keyboard strokes. This can be a legitimate software for monitoring, but it is mostly used for malicious purposes. The malicious keyloggers can send keystrokes back to hackers. The information stolen such as username and password for various accounts and banking details. The details are most often used for identity theft	F-secure, 2017
Ransomware	A type of malware which infects a computer system and limiting or preventing users from using it by encrypting it until a ransom is paid	Kaspersky, 2018
Botnets	Connected computers networks which have been infected by malicious software called bots. As a single entity, it is essentially a computer program which is trojan-like. As a connected network of bots, they can be remotely coordinated and synchronised by hackers. It can control all the infected computers to perform malicious deeds such as Distributed Denial of Service (DDoS) attack, large scale spam campaign, or any kinds of malicious purposes	Kaspersky, 2018

2.3 Purpose and Types of Malware Analysis

Malware analysis is the process of studying malicious program codes or behaviour executions of programs systematically. It is performed to discover the purpose of a given malware sample by identifying its characteristics, functionalities, risks, and intentions. Depending on the techniques and level of analysis, it can also be used to determine the sophistication level of the malware, to identify vulnerabilities, or even to be used to hunt hackers. The insights gained from analysis will give analysts information on how to react to threats or refine existing methods to defend against malware. According to Sikorski, M. (2012), from a technical standpoint, in order to discover what a malware sample is doing,

some of the most important basic questions which analysts should ask when performing analysis are:

- i. What are its host-based indicators and/or network-based indicators?
- ii. Does it have persistence mechanism?
- iii. When is the date of compilation and installation?
- iv. What language it is written in?
- v. Is the sample packed, obfuscated, or designed to thwart analysis?
- vi. Does it have any rootkit functionality?

Regardless the methods and tools used, these are the questions that analysts ought to answer. There are basically two broad categories for analysts to perform analysis, which are static methods and dynamic methods. Static analysis refers to all techniques that analyses programs without executing it, while dynamic analysis refers to all techniques that analyses programs by executing and monitoring it. Each category comes with its own strengths and limitations.

2.3.1 Static Analysis

Static analysis is done without executing potential malware binaries (Dinaburg, Royal, Sharif & Lee, 2008). It is an important technique for reverse engineering and analysing malware samples. This is due to the fact that malicious samples do not usually come with source codes, so analysts have to make do with analysing samples in binary format or various

other formats, such as Java. Basic static analysis includes retrieving hashes, which is in MD5 or SHA256, file header details, packer details, metadata such as creation date, strings in file, and embedded resources. When more details or advanced techniques are needed to decode malware, advanced static analysis techniques are performed. For cases like this, malware binaries are reverse engineered by means of disassembling. By decompiling binaries, analysts will gain insights into the working codes of the sample in the form of assembly language, as that is the highest-level language can be reliably and consistently reversed from binary codes without losing information (Sikorski, 2012). Logical structures, data flow, and data content are revealed in the process, helping analysts to make sense of the low-level representation of the decompiled binaries.

2.3.2 Dynamic Analysis

Dynamic analysis is performed by executing malware binaries in a controlled and isolated environment, such as virtual machines (VM); by using a debugger, or by running it in a specialised sandbox, since it has destructive and disruptive capabilities against the hosts and/or networks it resides in. There are various techniques to enable dynamic analysis, which includes function call monitoring, function parameter analysis, information flow tracking, instruction traces, and AutoStart extensibility points (Gandotra, Bansal & Sofat 2014). By performing dynamic analysis, analyst will be able to monitor the actions performed, and study the results of it. This includes files that are created, modified, or deleted, changes performed on the registry, process that are created, DLLs loaded before and during execution, network activities such as opened connections to the Internet and information which are downloaded or uploaded, and access to other OS services such as kernel drivers, installed services, and drive access.

2.3.3 Discussion on Malware Analysis Techniques

Static analysis is a powerful and useful technique. However, there are some limitations to it. Generally, malware analysis using static method is reduced to analysing binaries due to the unavailability of its source codes (Egele, Scholte, Kirda & Kruegel, 2012). Under normal conditions, this method is fast and reliable, but modern malware writers often employ techniques such as code and binary obfuscation, to resist reverse engineering or thwart analysis (Gandotra, Bansal & Sofat, 2014). Code obfuscation makes codes appear gibberish and it is designed to slow analysts down when analysing samples. Binary obfuscation employs method such as compression to change the structure of binaries to resist being disassembled. Due to this, disassembling obfuscated binaries will lead to inaccurate results due to obfuscators obscuring program flow (Aman, 2014). Polymorphic and metamorphic malware samples also pose problems as it is very hard to statically analyse malware that employs these techniques (Sihwail, Omar & Zainol Ariffin, 2018). Moreover, malware which relies on values that cannot be determined, such as indirect jump instruction or system date makes static analysis even harder.

Dynamic behaviour analysis on the other hand, shows promise in combating malware, especially unknown and complex malware. It allows analyst to reason about the execution as it allows precise analysis by showing useful indicators based on the runtime information (Aman, 2014), which are otherwise unknown by just analysing the disassembled codes. As a result, it cannot be defeated or obstructed by malware with mere obfuscation or polymorphism (Shahzad, Shahzad & Farooq, 2013). It also reveals the hidden nature of malware, as some malware will only reveal its hidden malicious nature when it is executed.

It could also be that the sample itself is benign/legitimate program but will download the actual malicious payload when executed.

The methods described in this section are the basic methods used by analysts to analyse samples. On its own, it is not enough to alleviate the threat of exponentially increasing threat of malware. The main reason for it being that it does not scale to the problem of big data efficiently.

Thus, the focus of this study will be an intelligent framework based on dynamic analysis, specifically based on automated behaviour analysis of malware to aid in quick analysis and identification of malware family and types. Dynamic analysis is chosen as the base analysis method due to the advantages it has. Machine learning is a perfect complement for this kind of task, as it is essentially a method of data analysis and automated model building for complex data (Shalev-Shwartz & Ben-David, 2016). It enables computers to find hidden insights or patterns from data and make predictions without being explicitly programmed. It has been proven to be a good complement to many problems, and which have been used in cyber security domains.

In the domain of malware analysis and detection, machine learning is commonly employed, and this is because of the ability for it to analyse and determine quickly large volume of malware samples daily. Amongst the most popular methods of machine learning used are based on supervised and unsupervised techniques, which encompasses pre-processing stage of data selection, data extraction, data representation, and also data classification and data clustering. The machine learning types which are focused for this research work are

explored, studied and reviewed, and divided into supervised and unsupervised learning methods.

We will study the application of machine learning methods in terms of malware analysis by comparing and reviewing related works in this chapter and discuss it by breaking down the basic components of it in the sections below by stating the definition and comparing related works.

2.4 Related Works

As malware threats becomes more dangerous by the day, there are many researches done on malware analysis techniques to counter the threats fast and efficiently using scalable methods. Automated malware analysis is very challenging, and there have been a lot of previous researches done on this issue. This chapter includes and discusses recent works on automatic malware analysis specifically in the area of data mining/machine learning utilizing supervised and unsupervised learning, while also discussing about feature selection, extraction, and representation.

2.4.1 Supervised Learning

Supervised learning are data mining or machine learning methods which utilises labels, and the algorithm is guided to what conclusions that it should come up with (Shalev-Shwartz & Ben-David, 2016). There are two categories of supervised learning, which are classification and regression. Classification outputs finite values while regression outputs continuous values (Murphy, 2013). Supervised learning requires the data to be labelled correctly and

that the possible outputs are already known beforehand. For example, a classification algorithm can be used to classify spams or non-spams, while regression can be used to predict prices of real estates.

In the malware analysis domain, it is one of the most used methods. Santos et al. (2013) uses a number of classification model, which are; Decision Trees (DT), K-Nearest Neighbour (K-NN), Bayesian Networks, and Support Vector Machine (SVM) in order to test which classification algorithms will work best for their dataset, for which they combined both static and dynamic features and chose the top 1000 static features and top 63 dynamic features embedded into vector space model. In a similar work where Galal et al. (2015) also utilised custom feature vector, by dividing API sequences into four categories; Handle creation, Handle-dependent, Handle release, and Simple, the authors use Decision Trees, Random Forest (RF), and SVM for classifying malware into families.

Shijo and Salim (2015) also created a custom feature vector which combines static and dynamic features; printable string information for static features, and API call sequence for dynamic features. They then used SVM and RF algorithms to classify the samples. Lin, et al. (2015) reduced the feature space using Principal Component Analysis (PCA) and Kernel Principal Component Analysis (KPCA) by optimising features which have been selected using TF-IDF, in order for the training time to be reduced as much as possible using online SVM. Pektaş and Acarman, (2018) used a single n-gram trie to map the information of all API call sequences, and the trie contains variable length n-gram to map behaviours accurately. They also use VirusTotal (*VirusTotal*, 2018) service to label samples, and their corresponding n-gram. Online algorithms are then used and evaluated in distributed computing environment, with Confidence Weighted (CW) and Adaptive Regularisation of

Weight (AROW) chosen. Offline algorithms; K-NN, RF, and SVM performance are also evaluated against the online algorithms. Moon, Im, Kim and Park (2015) applied Decision Tree algorithm in Intrusion Detection System (IDS) which analyses malicious behaviours by capturing session, protocol, DNS, IP data flows at network level and integrity, process, API, register, and driver data logs at system level.

2.4.2 Unsupervised Learning

Unsupervised learning are data mining or machine learning methods where the algorithms learn to infer patterns and processes without reference to labelled or known outcomes (Shalev-Shwartz & Ben-David, 2016). Therefore, there are no ground truth values or correct answers unlike in supervised learning. It is a very useful learning method to be used for discovering the underlying structure of the data or derive hidden insights out of data (Murphy, 2013).

Unsupervised learning methods are starting to gain popularity in the cybersecurity world as it allows insights into patterns of data. It can be used to detect anomalies and detect unknown or unique samples, although it is also a known issue that clustering results are hard to interpret due to the lack of data labels, and also methods to verify the data. Though it has some weaknesses as stated, it is an extremely useful learning method. Rieck et al. (2011) published a paper detailing a malware behaviour framework which is unsupervised but utilising both clustering and classification to achieve fast learning and processing speed for malware behaviour classification. The clustering algorithm used is Hierarchical Clustering, for which centroids are calculated to find the cluster centre when forming clusters. For classification, they utilised the centroid of each formed clusters as the prototype vectors to

represent formed clusters, in order to assign new known malware samples to existing clusters and assign unique or unknown malware samples into a new cluster.

In a work by Jang et al. (2011), they also utilised Hierarchical Clustering algorithm as the main clustering algorithm, to cluster malware static features or behaviour features which are represented as bitc vectors. They then applied Co-clustering algorithm to mine and discover inter-family and intra-family semantic similarities. Park and Reeves (2011) utilised a different methodology towards clustering malware, whereby they used a Graph algorithm, specifically Kernel Object Behavioural Graph (KOBG), to create a graph for each malware sample. KOBG works by creating a graph based on kernel objects, instead of API call sequences. They then cluster the malware graphs using Weighted Minimum Common Supergraph, where clusters are based on representative points, to achieve a family representation for each cluster formed.

Hidden Markov Model (HMM) is another algorithm which is popular in cybersecurity research, as it can learn the best representation of features, given the initial conditions. Annachhatreet et al. (2014) used HMM to train malware OpCode sequences in order to find the best representation for it. The scores obtained are normalised using log likelihood, to remove bias, as the initial data that are used to train have unequal length. In a related work which utilised HMM, Pai et al. (2016) trained HMM using OpCodes as well. The scores obtained from the training are also normalised. The difference is that while the former uses similarity in the scores to cluster malware, the latter uses K-Means and Expectation-Minimisation (EM) algorithms to cluster based on the scores obtained.

Hierarchical clustering is a very versatile algorithm, and it is also a simple one to use, due to the fact that a predefined cluster size need not be set before the clustering is done. There is however, an algorithm which is simpler in nature, which is called Disjoint-set Forest, and has almost the same properties as hierarchical clustering, and it takes lesser time to cluster. Oprişa, Cabău and Pal (2014) applied Disjoin-set Forest to a modified suffix tree which uses a custom distance measure named deq. It produces similar results to hierarchical clustering but takes lesser time to cluster.

Burnap, French, Turner and Jones (2018) used Self-Organising Map (SOM) to learn fuzzy boundaries from the data, from which features sets which are more flexible can be derived. The feature sets are then used for classification of the dataset. The authors use an ensemble of Logistic Regression (LR) to classify the data. Mohaisen, Alrawi and Mohaisen (2015) use a module called AutoMal to collect behavioural artifacts, and use another module called MaLabel, which uses those artefacts to create representative features using hierarchical clustering. The feature clusters, which are the representative features, are then trained using classifiers such as SVM, LR, Classification Trees, K-NN, and Perceptron.

2.4.3 Feature Selection, Extraction and Representation

Machine learning techniques rely on data to work. Most of the time, data for analysis are in improper or raw format, and thus unsuitable for machine learning algorithm to ingest. Therefore, the dataset must be cleaned up, structured, and formatted in such a way that the machine learning algorithm can make use of. This is true for all kinds of machine learning tasks whether supervised or unsupervised algorithms are used. For static malware analysis, the binaries contain certain data which are useful to the analysis, while for dynamic malware

analysis, the execution of the binaries produces important and useful behaviour logs, and these are called features. In this thesis we will review past works on dynamic malware analysis feature selection, extraction and representation.

Rieck, Trinius, Willems and Holz (2011) used a specially created metalanguage format called Malware Instruction Set (MIST) to convert behaviour logs into a format more suitable for machine learning usage. The format retains the sequence of API calls and also API call values. The API calls are coded based on categories and types of API calls, ranked based on importance. It is an efficient format due to the fact that it hashes the original data using non-cryptographic hash, then change the hash into hexadecimal representation, which reduces the size and complexity to manipulate and learn about the data. The new data format is then turned into a sliding window of n-shingles for processing using machine learning algorithm. Jang, Brumley and Venkataraman (2011) similarly used hashing technique to manipulate the malware behaviour logs, but they used feature hashing instead to turn original features in text format to hash values, then deriving a hash fingerprint out of it. This allows simple comparison of samples and achieves high processing throughput.

Park and Reeves (2011) uses a different technique to extract features out of behaviour logs. By using graph technique, kernel object behavioural graph for each sample are created to represent flow of kernel objects. Jang, Woo, Yun and Kim (2014) similarly used lightweight graph algorithm to build graph of system call dependency, using social network analysis tools. The graph created are then compared at the nodes with predefined malicious system call dictionary in order to refine the graph generated. The degree of distribution, degree centrality, and average distance are also measured. Wuechner, Cislak, Ochoa and Pretschner (2017) also used a variant of graph algorithm called compression-based graph mining to

create flow graph. It combines lazy graph mining technique, by integrating the graph creation with domain expert knowledge database, in order to create high quality graph representative of malware behaviour.

Santos, Devesa, Brezo, Nieves and Bringas (2013) uses both OpCodes and API call sequences to build a malware profile for each sample and used Information Gain to select 1000 static features and 63 dynamic features which are most important to each sample to build feature vector with. A similar work to this was done by Shijo and Salim (2015). Instead of using OpCodes for static features, they use printable strings. For dynamic features, they use API call sequence. The feature vector built upon combining static and dynamic features. The static feature portion of the feature vector are based on the frequency printable strings, while for dynamic features, it is based on n-grams of API call sequence. Another work which is based on dividing features based on categories is by Galal, Mahdy and Atiea (2015). This work is different due to the fact that they only utilise dynamic features, which are divided based on API function categories, which are; Handle creation, Handle-dependent, Handle release, and Simple. They refined the process to differentiate between malicious and benign samples using a global action-list.

Lin, Wang, Xiao and Eckert (2015) transform malware behaviour logs into n-gram, then perform TF-IDF weighting on it to select feature sets from prominent n-gram tokens. The outcome from selecting prominent n-gram feature sets based on the application of TF-IDF produces optimal accuracy in the results and shorter training time. The reduced feature sets are then converted from term frequency data by using Principal Component Analysis (PCA) and Kernel Principal Component Analysis (KPCA) algorithms to determine an optimal reduced feature set that yields high accuracy and short training time. In the work of Pektaş

and Acarman (2018), the authors use variable length n-gram to represent API call sequences in order to capture more behaviour sequence, and the API call sequences which are of variable n-gram are stored in a single trie.

Annachhatre, Austin and Stamp (2014) explore the use of Hidden Markov model to learn features which are useful for malware sample characterisation. After multiple Hidden Markov Models are trained based on a given observation sequence, malware samples are then clustered based on the scores obtained which is log likelihood format. A similar work was done by Pai, Troia, Visaggio, Austin and Stamp (2016) using Hidden Markov Model as well, where the models are trained using the extracted sequence, and the scores which are generated are normalised in the form of log likelihood to remove bias.

There are also works which explores the usage neural networks or deep learning to learn feature representation. Wang and Yiu (2016) takes raw malware API call sequences and feeds it into recurrent neural network-based Autoencoder (RNN-AE) in order to learn low dimensional representation of malware of the raw API call sequence. Kolosnjaji, Zarras, Webster and Eckert (2016) on the other hand, clean up the behaviour reports, then turn the report in n-gram format to model the API sequence. The data is the feed to a stacked convolutional neural network and recurrent neural network, in order to get a hierarchical feature extraction architecture. Huang and Stokes (2016) uses API call sequence and parameters to represent API events using a many-to-one mapping during the feature selection stage. The reason behind this mapping concept is because there are multiple APIs which can be used to achieve the same objective, thus by mapping it to a generalised API name, it is simpler to model API sequence similarity. After the APIs are mapped, the API sequence are turned into n-gram format, in which for this work, the authors use 3-gram. The resulting

feature space from this is large, therefore to reduce it the authors use mutual information to generate features that best characterise each. However, the resulting feature space is still large, so the authors further reduce the feature space by using random projection algorithm. The data are then fed into a standard feed forward neural network to learn the best feature representation.

2.4.4 Malware Labelling

Malware samples are given labels in order to identify its type, variant, platform it is affecting, and programming language it is written in. There is a naming standard for labelling malware samples proposed by Computer Antivirus Research Organization (CARO) in 1991 (CARO, 2018), which attempts to standardise malware naming across the industry. However, most cybersecurity firms have their own naming standard, which could be due to how the malware was analysed and grouped. Take for example the botnet Zeus, it has different names depending on the company which identifies it;

- a. F-Secure - W32/Zbot
- b. Symantec - Trojan.Zbot
- c. McAfee - PWS-Zbot

Labelling can be done manually or automatically. Manually labelling malware samples is clearly not feasible today due to the large number of samples in the wild. Therefore, many researchers have come up with automatic malware classification using machine learning, however it is still a problem as threats escalate over the years (Dahl, Stokes, Deng & Yu,

2013). Consensus of varied labelling from multiple vendors, ranking, or techniques based on quantitative analysis can be done to standardise labelling (Perdisci & U, 2012; Sebastián, Rivera, Kotzias & Caballero, 2016). Some framework, such as the one researched by Wang, Hu, Meng and Sailer (2014), even delves into reconciling malware labels without changing the labels given by vendors and can be used to query multiple labels to check for sample similarity. This is useful as it can even be used for threat intelligence across security vendors and organisations. However, for this thesis, we will only do basic labelling, by choosing labels from selected Antivirus (AV) vendors, and for which the labels have similar structures, in order to streamline sample labelling. The standardisation of labelling will not be delved into as it is a big research on its own.

2.5 Discussion

In this section, we will discuss upon the related works, by discussing the strengths and weaknesses of those works and the main focus will be on the techniques used by these works and how it can be applied towards analysis rather than detection as shown in Table 2.2, and a summary of how it leads towards the proposed framework for this thesis.

Table 2.2: Comparison of related works

Year	Authors	Approach	Strength	Weakness
2011	Rieck, Trinius, Willems, & Holz	Combining clustering and classification for quick learning to find novel or unique samples	Fast clustering time due to the usage of centroid-based clustering, and using prototype vectors as representative of clusters to classify malware samples fast	Centroid-based clustering has inherent weakness as inversions can occur, whereby joined pair are more similar to the parent cluster than to each other

Table 2.2 continued

2011	Jang, Brumley & Venkataraman	Using feature hashing to represent samples as bitvector signature, and using co-clustering to mine correlated inter-family and intra-family semantic similarities	Very fast clustering time, and able to mine correlated semantic similarities	Feature hashing destroys original data as hashing is a one way format
2011	Park & Reeves	Using Kernel object sequence graph instead of API call sequence graph	Low false positives, and robust against system call attacks	Building a kernel object graph takes a lot of time, and deriving a supergraph out of similar sample clusters takes a lot of time
2012	Santos, Devesa, Brezo, Nieves & Bringas	Combines both static and dynamic features and using Information Gain to select top features	Combining static and dynamic features achieve slightly higher accuracy rate in classifying malware samples	Restricting features to just top 1000 for static, and top 63 for dynamic will yield poor classification performance in regards to unknown malware as it may have unknown features
2013	Annachhatre, Austin & Stamp	Using Hidden Markov model to learn feature representation	The HMM model analysis can possibly detect previously unseen malware	HMM takes a long time to be trained, and hard to understand

Table 2.2 continued

2014	Galal, Mahdy & Atiea	Using behaviour-based features to detect malware	API call sequence can reveal intentions better than using static features, and by referring malware API sequences of benign programs, it can filter and lower false negatives	Classification cannot reveal or detect unknown malware samples
2014	Oprişa, Cabău & Sebestyen Pal	Using suffix tree with custom deq distance measure, and using Disjoint-set forest clustering algorithm	Disjoint-set forest is a faster and simpler alternative to cluster as compared to Hierarchical clustering	Suffix trees takes time to build, despite deq distance only allowing insertions and deletions, as compared to Levenshtein distance which includes replacement
2014	Jang, Woo, Yun & Kim	Using lightweight graph algorithm analogous to the ones used for social network analysis	Can differentiate malicious and benign programs well, classify malware using fewer features, and learn features more accurately using social network analysis	It is very slow to derive results using graph method, and it is counterproductive although the results are very accurate
2015	Pai, Troia, Visaggio, Austin & Stamp	Using Hidden Markov model to train samples, and using Expectation-Maximization to cluster samples	Clustering can learn about unknown or novel patterns and detect new malware	Expectation-maximization may be very slow to converge and produce results, especially if the dataset is large
2015	Shijo & Salim	Combining static features (printable strings information) and dynamic features (API call sequence)	Adding static features increases the accuracy by almost 1.5% by using SVM	Classification cannot detect unknown or novel malware samples

Table 2.2 continued

2015	Mohaisen, Alrawi & Mohaisen	Using behaviour-based analysis for better malware analysis and classification, by collecting feature artefacts, and create representative features out of it for building classifiers	By learning feature artefacts using clustering, the classification can be more accurate	Classification cannot detect new or novel samples
2015	Lin, Wang, Xiao & Eckert	Using TF-IDF to select important features, and applying PCA or KPCA to further reduce the feature space for online SVM learning	Reduced feature space decreases learning time	PCA is fast for reducing feature space, although it is not as good as KPCA which is slower in reducing feature space, and online learning is hard to tune as the weights will change over time as new data comes in
2016	Stokes, Chandola, Seifert & Burrell	Using a custom similarity measure, Weighted Square Match (WSM) and its unweighted variant to measure similarity	WSM is robust to mimicry and NOOP evasion techniques Using MinHash to reduce the number of pairs for comparison speeds up computation	The system can detect APT, but it isn't top performing, as clustering is not applied to allow discovery of APT variant or novel APT detection
2016	Wang, & Yiu	Summarizing malware API call sequence using seq2seq learning and FAP generator and applying multi-task malware learning model using RNN-AE, and a custom	Automatic feature representation and feature interpretability using automatic feature summarization for API call sequence	Deep learning model is very resource intensive and slow to train

Table 2.2 continued

		decoder to summarize API call sequence		
2016	. Kolosnjaji, Zarras, Webster & Eckert	Using deep learning by combining convolutional and recurrent neural network in hierarchical manner	Combining convolutional and recurrent neural network architectures gives better results than using either one of the neural networks, HMM, or SVM	Training time for deep learning architecture is very long, which is about 3-10 hours for this research
2016	Huang & Stokes	Using deep multi-task malware classification model which are trained using data extracted from dynamic analysis	The proposed model leads to modest gains when compared to shallow neural network architectures	Including dropout increases accuracy and also reduces the training time, but it is still long, as compared to non-neural networks algorithms
2017	Wuechner, Cislak, Ochoa & Pretschner	Using compression-based mining on qualitative data flow graphs to derive highly accurate models	Using compression-based graph mining for behaviour-based malware detection to achieve more accurate detection than graph generated with frequency-based graph mining	This research incorporates domain expertise, which complicates automated learning, and graph algorithms are complex to train
2017	Pektaş & Acarman	Using variable length n-gram to cover more malicious behaviours	Using variable length n-gram makes detection rate higher as it can cover more malicious behaviours	The advantage from using variable length n-gram is not that warranted as it is time consuming to extract variable length n-gram

Table 2.2 continued

2017	Moon, Im, Kim & Park	Using decision tree to classify malicious behaviour in the network and system level	It is easy to interpret results by using Decision Tree algorithm	Classification algorithm is not designed to discover unknown or novel samples
2018	Burnap, French, Turner & Jones	Using self- organising feature maps to generalise across different datasets and capable of online learning	Using SOFM to produce a probabilistic model, to be used as a new feature set for classification yields very good results	It takes time for SOFM to learn feature sets as it is a hill climbing type algorithm, and it is hard to interpret the results

From the comparison of approach, strengths and weaknesses in Table 2.2, the main focus of machine learning approaches is clearly in the post processing stage, where the features and format of the data are the concern, and the learning algorithm, where decision is made on whether to use supervised or unsupervised learning algorithms, to cluster or classify the dataset. The work of Rieck et al. (2011) is novel as they have developed a special metalanguage for malware behaviour, and it is a good way to standardize malware behaviour information. Based on literature on malware metalanguage, there are no literature showing methods of standardizing malware analysis report output previously, and no proper focus on the conversion to machine efficient metalanguage, hence the creation of the metalanguage format Malware Instruction Set (MIST) to overcome this limitation. MIST will be studied and evaluated for usage for this study.

Oprişa, et al. (2014) has also introduced the usage of an interesting algorithm called Disjoint-set Forest, which could possibly produce good results even in distributed computing

environment, and with an efficiency higher than Hierarchical clustering. Disjoint-set Forest is an efficient algorithm based on Kruskal algorithm, which is a very simple graph algorithm. It speeds up cluster computation by computing based on representative set rather than performing all pair analysis. This is an advantage because it mitigates the common problem on cluster computation time. It can also be adapted to parallel computing, which can speed up cluster computation time even further. Disjoint-set Forest algorithm will be studied and evaluated in this research.

However, there are also some of shortcomings of the reviewed literatures. One specific shortcoming is the lack of focus on Nearest Neighbour Search (NNS). The argument here is that there should be a focus on fast and scalable algorithms used for NNS for unsupervised learning algorithms. Therefore, fast nearest neighbour search algorithms will be studied and implemented to overcome the processing speed limitations faced by the works reviewed. The focus of this thesis is also on efficient feature selection and representation, efficient nearest neighbour search algorithm and machine learning algorithms. From this discussion, the framework will be designed in Chapter 3.

2.6 Summary

This chapter has provided explanations on the fundamentals of malware and malware analysis. It discussed the differences between static and dynamic analysis methods. Based on the advantages of dynamic analysis against modern malware, it was chosen as the base method for this research, specifically sandbox analysis using API hooking and DLL injection. We have also discussed the shortcomings of dynamic analysis methods with the massive daily influx of malware, thus the need for automated analysis using data mining and

machine learning approaches to alleviate this problem. Apart from that, in this chapter we have also delved into the basics of machine learning components, and we discussed it in the context of malware analysis. In the related works section, recent works on malware analysis using machine learning approaches are reviewed and discussed. By understanding the advantages and shortcomings of those works, an experimental framework to improve upon the weaknesses of past works is proposed. In the next chapter, the methodology and proposed framework will be detailed.

CHAPTER 3

METHODOLOGY AND FRAMEWORK

3.1 Introduction

In this chapter, the design of the proposed framework and methodology are explained in detail. Firstly, the general flow of the framework is shown, then each components of the framework are explained in detail. The contribution of this research work is that it uses hash fingerprints to represent samples, utilises a fast nearest neighbour search, and uses a very simple yet effective and accurate clustering algorithm. The selection of algorithms used in this framework is based on studying related works. As a result, this framework would be able to identify malware by family, and unknown malware. It would first perform sandbox analysis on the selected samples. Then, behaviour reports are pre-processed, and only features which are useful for malware analysis are selected. The selected features are then transformed into a metalanguage format which is suitable for machine learning before clustering is performed. A fine-grained check will be done once initial clusters are formed, to remove false positive and false negative data from the clusters, and it will be repositioned into the correct clusters. If there are no cluster match for the data, then the data will be placed in a cluster for rejected samples. This proposed method is called MalSketch.

3.2 MalSketch Overview

In this subsection, the overview of the MalSketch is shown in Figure 3.1. It shows the general flow to analyse malware behaviours. The process starts by converting the behaviour reports into a metalanguage format known as Malware Instruction Set (MIST) specifically, which is

suitable for machine learning usage, then converting the converted behaviour reports into unique fingerprints using Minhash algorithm. Then, a fast nearest neighbour search algorithm called Locality Sensitive Hashing (LSH) is used to search and group similar samples together. After that, a simple but efficient clustering algorithm is applied to the nearest neighbour search results to cluster data together.

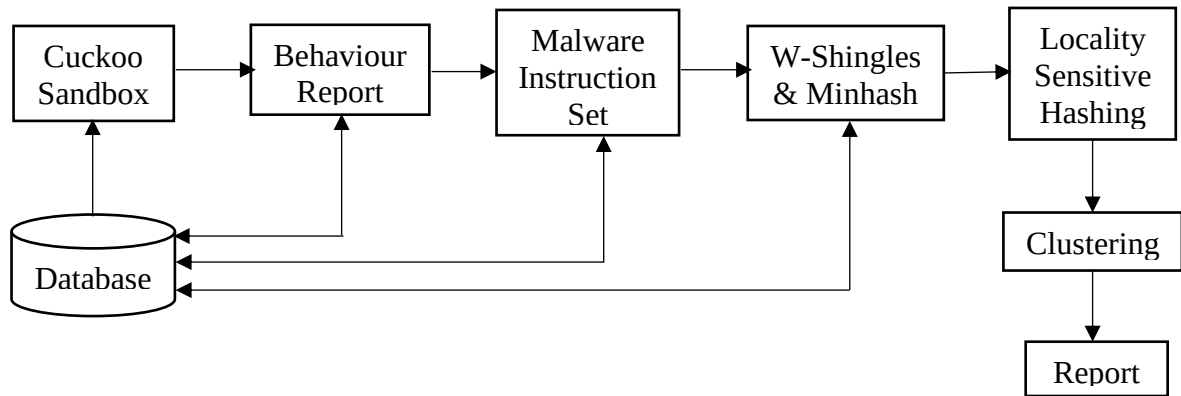


Figure 3.1: MalSketch Overview

3.3 Framework Components

In this section, each component of the framework is explained in detail. The main components or modules consist of the database module, sandbox module which includes the behaviour reporting module, Malware Instruction Set (MIST) module which is the metalanguage format suitable for machine learning, Minhash module, Locality Sensitive Hashing module, and clustering module.

3.3.1 Database

This subsection discusses the flow of the framework's database module. It encompasses the stored behaviour report behaviour reports from Cuckoo Sandbox, the storage of the transformed report into Malware Instruction Set (MIST) to the storage of Minhash values of the MIST reports.

For this research which deals with a large number of malware samples numbering at 65,000, quick access and systematic storage system is a prerequisite. Keeping track of unique instances of each file, accessing and storing the malware behaviour reports from sandbox analysis and Malware instruction Set (MIST) reports, as well as Minhash values for each MIST report quickly requires a good database system. The choice of database to use is critical as it will influence the operation of the framework. Since the nature of the data that is dealt with in this research are text files, a good choice for database type is NoSQL, specifically the document store variant. The chosen document NoSQL database platform chosen is MongoDB. MongoDB is chosen because it can store the data in a very rich way while being capable of holding arrays and other documents. It is also sensitive to workloads and can work well with large databases. The Data Flow Diagram (DFD) of the database is shown in Figure 3.2.

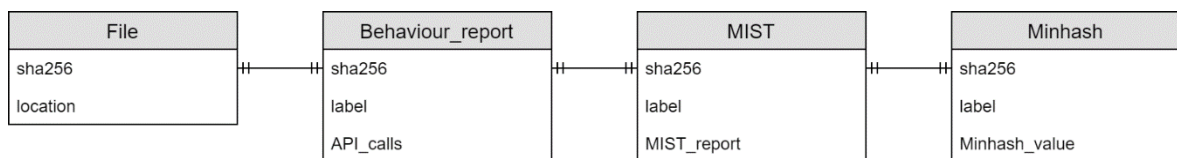


Figure 3.2: ERD of the framework database

File represents a malware sample and the attributes are hash in SHA256 and its file location. A Behaviour_report is a behaviour report which is produced by Cuckoo Sandbox, and it contains SHA256 hash, label, and the API calls, which is the report itself. MIST report is a behaviour report format which has been converted into a machine learning suitable format. It contains SHA256 hash, label, and mist_report is the actual report itself. MIST will be explained in more details in Subsection 3.3.3. Minhash represents the signature of malware samples derived from the MIST reports. Its attributes are SHA256, label, and Minhash_value. It will be explained in more details in Subsection 3.3.4. A malware sample produces a Behaviour_report. A Behaviour_report produces a MIST report, and a MIST report produces a Minhash.

3.3.2 Sandbox

A sandbox is a separated and contained environment, usually used to execute untrusted or untested programs, without risking the security and stability of the host machine or host operating system. It is a special instance of virtualisation. In this research, the focus is on dynamic malware analysis, therefore the sandbox used is a specially designed sandbox for capturing malware behaviour at host and network levels. For this research Cuckoo Sandbox is chosen as the sandbox framework to be used for malware behaviour analysis. Cuckoo Sandbox is a very versatile sandbox framework which can identify, execute, and monitor many formats of files and collect comprehensive information about what a malware does while executing in an isolated environment (Cuckoo Sandbox, 2018). It can collect API call sequences of class being performed, file created, deleted, or downloaded, collect memory dumps, collect network traffic, take screenshots, and collect full dumps of malware behaviour. Figure 3.3 shows the overview of a single instance of Cuckoo Sandbox.

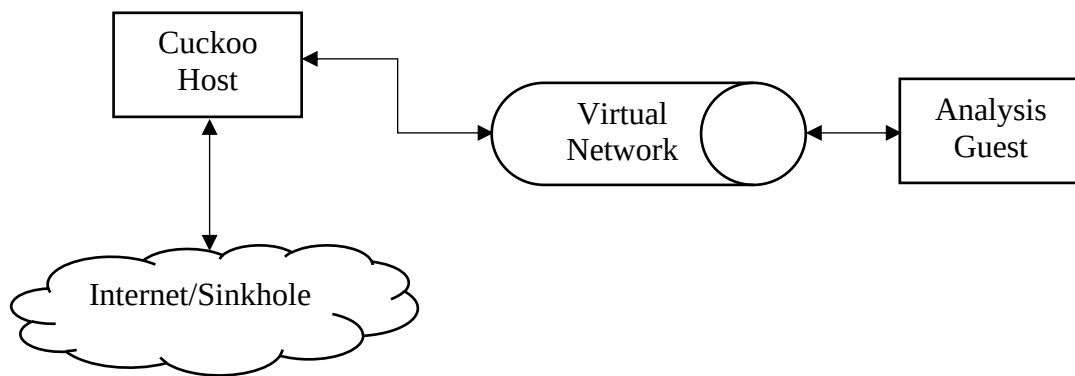


Figure 3.3: Overview of single instance Cuckoo Sandbox

- Cuckoo Host represents the host environment where Linux operating system is used, for its robust and secure environment. The host environment is responsible for the Guest operating system which is installed in a virtual environment, and it is also responsible for the analysis management, for example, analysis queue, start, stop, and generating reports.
- The Analysis Guest represents the guest environment where Windows operating system is used, is a virtualised environment where the analysis happens. It is a clean but unsecured environment to run samples. The analysis actions in this space is reported back to Cuckoo Host.
- Virtual network is an isolated network environment which is used by the virtual machines to communicate through the Host OS to send and receive data.

The single Cuckoo Sandbox instance analysis is set at three minutes per analysis. With a large number of samples, this would not be feasible as it is too time consuming. Therefore, multiple instances of Cuckoo Sandbox in a single host are used utilising a distributed

computing methodology. Figure 3.4 shows the overview architecture of the distributed Cuckoo Sandbox setup.

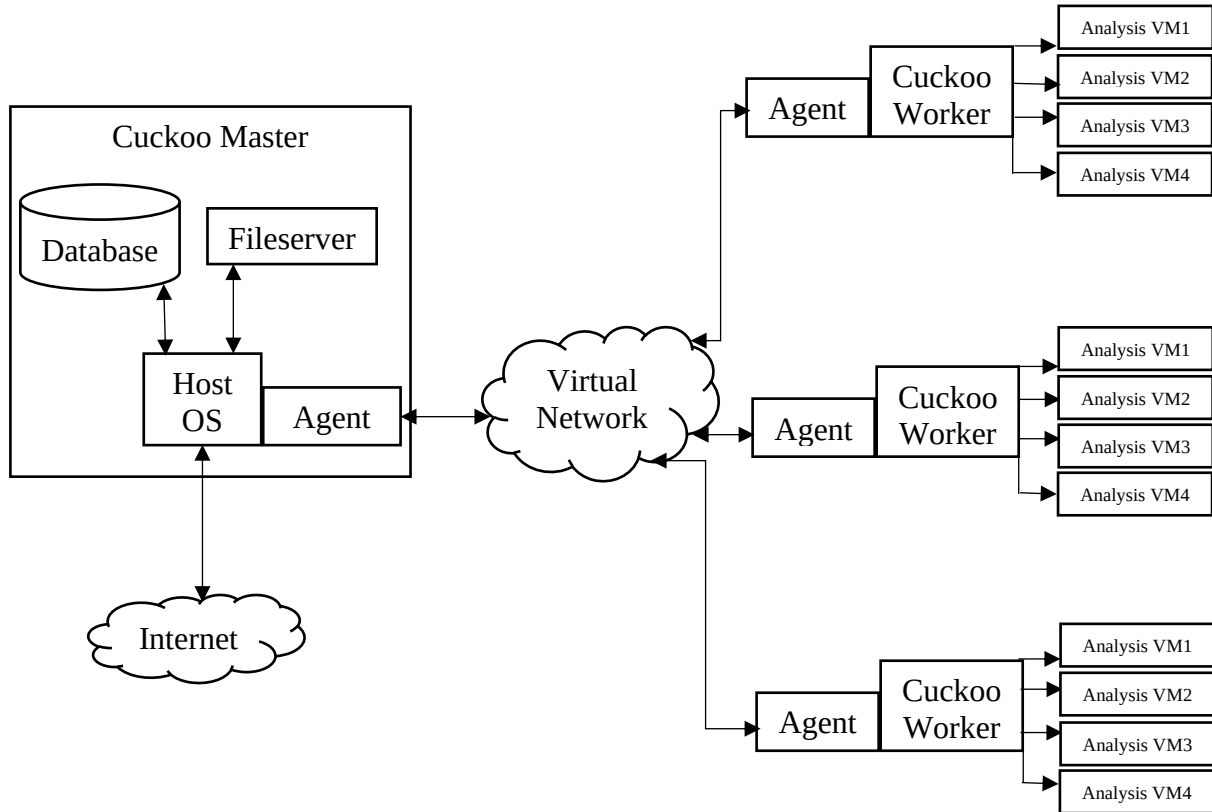


Figure 3.4: Overview architecture of distributed Cuckoo Sandbox analysis

For this research, the distributed Cuckoo Sandbox analysis environment is designed based on Figure 3.4. The flow of the sandbox is described below.

- Cuckoo Master contains database which stores samples according to SHA256 hash. The database design and flow are described in subsection 3.3.1. Fileserver is used to communicate and view analysis progress on Cuckoo Workers. Meanwhile, Host OS is where the Fileserver is hosted, and it is where the Cuckoo Agent passes messages to and from Cuckoo Workers.

- Virtual network is used to map each Cuckoo Workers to the Cuckoo Master, and it is used for communication purposes to and from the Internet as well, through Cuckoo Master Host OS. All communication from Host OS to Cuckoo Workers and vice versa are through this virtual network. All the behaviour reports from Cuckoo sandbox instances in the Cuckoo Workers are stored in the database which is on the Host OS.
- For every Cuckoo Worker, which is a host in its own right, it hosts four virtual machines for each contains a Cuckoo Sandbox instance, which works in headless mode (without GUI), in order to save RAM resources. Cuckoo Worker does not work on its own although it is a host, it gets analysis request from the Cuckoo Master, which queues and balances analysis loads for each Cuckoo Worker through Cuckoo Agent. The results for each analysis are stored in each Cuckoo Worker storage.

After the samples are executed in the sandboxes, the labels will be evaluated. By retrieving the labels from VirusTotal (*VirusTotal*, 2018), a list of labels from different vendors are listed in the behaviour reports. In order to cover all the samples with labels, vendors that have a large database of malware samples are chosen. For this experiment, the vendor from which labels are taken are:

1. F-Secure
2. Bitdefender
3. Microsoft
4. Kaspersky
5. Avira

6. AVG
7. ESET-NOD32
8. Avast
9. Trend Micro
10. McAfee

The labels are chosen in descending order of importance, as it is noted that the vendors with higher precedence in the list have naming convention which is very similar or the same as CARO naming convention (CARO, 2018). Figure 3.5 shows the pseudocode for the Labelling algorithm. A label will be chosen if it exists, based on the default label, F-Secure. If it does not exist, the sample will be compared iteratively until a label can be found. If a label cannot be obtained from the list, then SHA256 will be used to represent the sample.

```

Start

Set vendor list to ["F-Secure", "BitDefender", "Microsoft",
"Kaspersky", "Avira", "AVG", "ESET-NOD32", "Avast", "TrendMicro",
"McAfee"]

Select report folder

for every file
    Load report
        if behaviour process exist
            Select vendor according to list
            Rename file
        if behaviour process does not exist
            Use SHA256
            Rename file
        else
            error: not renamed
    end for
End

```

Figure 3.5: Pseudocode for labelling samples

After the labels have been defined, the behaviour reports will be saved into the database. Although the behaviour report can be utilised for machine learning purposes, it is not designed nor optimised for it. As shown in Figure 3.6, the format of the original behaviour report which is in JSON format is hierarchical in structure, and it is hard for it to be utilised efficiently for machine learning as it is not structured and optimised for it.

```

{
  "category": "registry",
  "status": "SUCCESS",
  "return": "0x00000000",
  "timestamp": "2012-11-06 20:39:30,274",
  "thread_id": "1672",
  "repeated": 0,
  "api": "RegOpenKeyExW",
  "arguments": [
    {
      "name": "Registry",
      "value": "0x80000003"
    },
    {
      "name": "SubKey",
      "value": "S-1-5-21-1547161642-1606980848-1708537768-500"
    },
    {
      "name": "Handle",
      "value": "0x00000170"
    }
  ]
},
{
  "category": "filesystem",
  "status": "SUCCESS",
  "return": "0x00000000",
  "timestamp": "2012-11-06 20:39:30,274",
  "thread_id": "1672",
  "repeated": 0,
  "api": "NtCreateFile",
  "arguments": [
    {
      "name": "FileHandle",
      "value": "0x00000178"
    }
  ],

```

Figure 3.6: Snippet of the original behaviour report

In the next subsection, a more suitable and optimised format called Malware Instruction Set (MIST) is discussed detailing its advantages and usage.

3.3.3 Malware Instruction Set

Malware Instruction Set (MIST) is originally written by Trinius, Willems, Holz, & Rieck (2009). It is a metalanguage format that is designed for efficient machine learning algorithm performance in terms of processing speed and accuracy by using approximation method by hashing the input. Input file can be file opcodes or behaviour report from execution of malware sample in a sandbox which captures its input and output through API calls at system and kernel level. After studying the literatures, it is found that almost no previous works focused on a proper format for malware behaviour report which is suitable for efficient machine learning usage. MIST is the first framework, as per literature reviews, that creates a proper metalanguage format for standardising malware opcodes or behaviour reports. It is not a new work, but because of its versatility, and it works so well, it is chosen for this research.

MIST has a number of advantages. Firstly, it standardises the formatting of the behaviour report. Secondly, it describes malware behaviour in sequence of instructions, and individual execution flows of threads and processes are grouped in a single report. Thirdly, it allows rearrangement of API call arguments, thus allowing prioritisation of arguments based on level of importance.

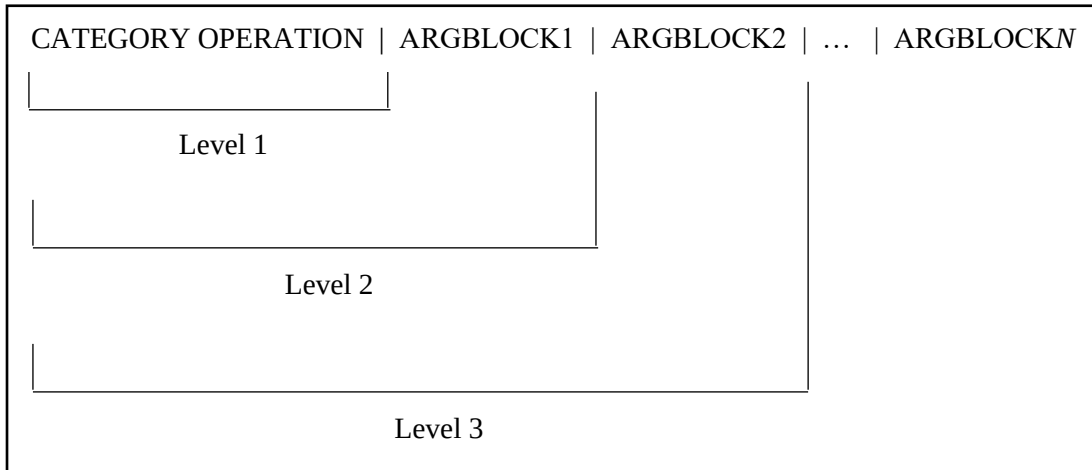


Figure 3.7: MIST schematic format

Figure 3.7 shows the schematic format for MIST. For each level of MIST instructions, they are separated by a block separator as indicated by a vertical bar. Category Operation is Level 1 of the instructions. It corresponds to the Windows API category and the monitored system call. These are codified in hexadecimal, for ease of understanding the codes, and also for simplifying learning of the dataset. From Level 2 onwards which contains ARGBLOCK (argument block), non-cryptographic hash is used. The original author uses ELF hash, but for this research work Murmurhash3 which is shown in Figure 3.8 is used instead, because it has better properties than ELF hash, and it is also faster, and more collision resistant.

The data for ARGBLOCK are system call arguments for which its MIST categories, encoding as well as the number of system calls for each category are shown in Table 3.1. This is defined by studying Cuckoo Sandbox documentations of system calls which are supported for monitoring by Cuckoo Sandbox as they are deemed important from malware analysis perspective, where 12 main categories are recorded. Although MIST is versatile and can be adapted to various other use cases, in this case it is designed to use the output of

system calls monitored by Cuckoo Sandbox thus the rationale behind designing it to support Cuckoo Sandbox behaviour report.

The system call arguments can be arranged according to the importance it brings for the analysis of malware behaviour, as shown in Figure 3.7. For example, in category 06 04, which is the System category with the system call being LdrGetDllHandle, the arguments which are important is FileName, which is set at level 2 of ARGBLOCK, as it can reveal the files which was created, deleted, or downloaded by malware, and usually malware from the same family will have the same filenames. Meanwhile, ModuleHandle is set at level 3 as it is not so important to the analysis of malware behaviour, as ModuleHandle ID can change each time it is called. Figure 3.9 shows the MIST format of the behaviour report after conversion.

Table 3.1: MIST categories, encoding number, and unique operations for each category

Encoding	Category	No. of system calls
01	Hooking	2
02	Network	20
03	Windows	3
04	Threading	12
05	Process	21
06	System	9
07	Services	6
08	Synchronization	5
09	Registry	21
0a	Filesystem	13
0b	Device	2
0c	Misc	1

```

Murmur3_32(key, len, seed)

2^{32} arithmetic.

c1 ← 0xcc9e2d51
c2 ← 0x1b873593
r1 ← 15
r2 ← 13
m ← 5
n ← 0xe6546b64

hash ← seed

for each fourByteChunk of key
    k ← fourByteChunk

    k ← k × c1
    k ← (k ROL r1)
    k ← k × c2

    hash ← hash XOR k
    hash ← (hash ROL r2)
    hash ← hash × m + n

with any remainingBytesInKey
    remainingBytes ← SwapToLittleEndian(remainingBytesInKey)
    remainingBytes ← remainingBytes × c1
    remainingBytes ← (remainingBytes ROL r1)
    remainingBytes ← remainingBytes × c2

    hash ← hash XOR remainingBytes

hash ← hash XOR len

hash ← hash XOR (hash >> 16)
hash ← hash × 0x85ebca6b
hash ← hash XOR (hash >> 13)
hash ← hash × 0xc2b2ae35
hash ← hash XOR (hash >> 16)

```

Figure 3.8: Pseudocode for Murmurhash3 32bit

```

06 04 | 0191feb5 00000000 | 03ad91e3 00000000 | 7c800000
06 06 | 0191feb5 00000000 | 0270a6b7 00000000 | 0012f568 77dd0000
06 06 | 0191feb5 00000000 | 0445cc8f 00000000 | 0012f8e8 7e410000
06 06 | 0191feb5 00000000 | 026425f0 00000000 | 0012f8e0 771b0000
06 06 | 0191feb5 00000000 | 01bd1e73 00000000 | 0012fba8 7c900000
06 05 | 00000000 0187706e 05448ca9 00000000 | 00000000 00000000 | 00000000 7c900000
06 06 | 0191feb5 00000000 | 01bd1e73 00000000 | 0012fba8 7c900000
06 05 | 00000000 0187706e 0100fb94 00000000 | 00000000 00000000 | 00000000 7c900000
06 06 | 0191feb5 00000000 | 00e398b9 00000000 | 0012fba8 77e70000
06 06 | 0191feb5 00000000 | 03ad91e3 00000000 | 0012fba8 7c800000
06 06 | 0191feb5 00000000 | 01bd1e73 00000000 | 0012fb70 7c900000
06 05 | 00000000 0187706e 027c09d6 00000000 | 00000000 00000000 | 00000000 7c900000
05 09 | 00000040 | 0001f89a | 0015f148 ffffffff
05 09 | 00000040 | 0000599b | 00189350 ffffffff
05 09 | 00000040 | 0000494f | 00183008 ffffffff
06 06 | 0191feb5 00000000 | 01bd1e73 00000000 | 0012fb14 7c900000
06 05 | 00000000 0187706e 0239efa5 00000000 | 00000000 00000000 | 00000000 7c900000
06 06 | 0191feb5 00000000 | 01bd1e73 00000000 | 0012fb14 7c900000
06 05 | 00000000 0187706e 012e823c 00000000 | 00000000 00000000 | 00000000 7c900000
06 06 | 0191feb5 00000000 | 01bd1e73 00000000 | 0012fb14 7c900000
06 05 | 00000000 0187706e 03bae428 00000000 | 00000000 00000000 | 00000000 7c900000
06 06 | 0191feb5 00000000 | 01bd1e73 00000000 | 0012fb14 7c900000
06 05 | 00000000 0187706e 04ed3472 00000000 | 00000000 00000000 | 00000000 7c900000
06 06 | 0191feb5 00000000 | 01bd1e73 00000000 | 0012fb14 7c900000
06 05 | 00000000 0187706e 00c1bbc5 00000000 | 00000000 00000000 | 00000000 7c900000
06 06 | 0191feb5 00000000 | 01bd1e73 00000000 | 0012fb14 7c900000
06 05 | 00000000 0187706e 022c5dff 00000000 | 00000000 00000000 | 00000000 7c900000
06 06 | 0191feb5 00000000 | 01bd1e73 00000000 | 0012fb14 7c900000
06 05 | 00000000 0187706e 00edb1b6 00000000 | 00000000 00000000 | 00000000 7c900000
06 06 | 0191feb5 00000000 | 01bd1e73 00000000 | 0012fb14 7c900000
06 05 | 00000000 0187706e 0272b6fb 00000000 | 00000000 00000000 | 00000000 7c900000
06 06 | 0191feb5 00000000 | 01bd1e73 00000000 | 0012fb14 7c900000
06 05 | 00000000 0187706e 023eacda 00000000 | 00000000 00000000 | 00000000 7c900000
06 06 | 0191feb5 00000000 | 01bd1e73 00000000 | 0012fb14 7c900000
06 05 | 00000000 0187706e 00ee222f 00000000 | 00000000 00000000 | 00000000 7c900000
06 04 | 0191feb5 00000000 | 03ad91e3 00000000 | 7c800000

```

Figure 3.9: Snippet of MIST report format

The link to the complete system calls category and system calls in each categories list can be found in APPEND section at the back of the thesis. The work of this custom MIST framework has been published on GitHub: <https://github.com/Navein/Cuckoo2MIST>.

3.3.4 W-Shingles and Minhash

After the behaviour reports has been converted into MIST format, W-shingles algorithm is used to transform the data for further processing. W-shingles or overlapping n-gram, commonly used in Natural Language Processing (NLP), is a technique used to tokenize words into overlapping contiguous sequence. For example, the word “Malware” can be tokenised in 2-gram as; “Ma, al, lw, wa, ar, re”. It is commonly used for set similarity due to the fact that fine grained comparison of words yields more accurate results as compared to comparing whole words or sentences as a set. It has proven successful in applications such as nearest-neighbour search, clustering, and code reuse detection. For this research, malware behaviour reports which have been converted into MIST format will be converted into w-shingles. Set comparison is done by comparing two sets of documents which have been w-shingled and the algorithm for it is called Jaccard similarity coefficient. It is defined below as;

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}, \text{ where } 0 \leq J(A, B) \leq 1 \quad (3.1)$$

where A and B represents sample sets.

However, by converting MIST reports into w-shingles, the feature space becomes larger, although it becomes more meaningful as the reports are tokenised to allow more accurate comparison between two documents. Due to the high dimensionality of tokenised MIST reports, it is not feasible to calculate similarity due to resource constraints, and also time constraints.

To overcome the problem of large feature space and to increase the efficiency of clustering, a sketch of the MIST report can be used as representative. Minhash was introduced by Broder (1997), and it is a hashing technique of representing data with a hash sketch, and it is used for quickly estimating the similarity of two sets. Therefore, Minhash is a good method of accurately estimating a sketch of how the original data would be.

Minhash approximates Jaccard similarity by using random sampling of elements in a set. Let h be a hash function, and $h(a)$ is used to map any elements in Set A to a distinct integer which is drawn from a linear distribution. For any two sets, A and B, Jaccard similarity can be expressed using the probability of hash collisions.

$$J_d(A, B) = 1 - P_r \left[\min_{a \in A} h(a) = \min_{b \in B} h(b) \right] = P_r \left[\min_{a \in A} h(a) \neq \min_{b \in B} h(b) \right] \quad (3.2)$$

Where P_r represents the probability of Minhash, when r is 1 when $\min h(a) = \min (b)$, and 0 otherwise.

Minhash has two variants, which are the single hash function variant, and also the multiple hash functions variant. This research objective is to create an efficient malware analysis framework in terms of speed and storage space used while still retaining a high degree of clustering accuracy, therefore the multiple hash function variants will not be looked into, as it is not space efficient, and also computationally expensive. Single hash function Minhash can be explained as; Let h be a hash function and let k be a fixed integer. If S is any set of k or more values in h , define $h_{(k)}(S)$ to be the subset of the k members of S that have the

smallest values of h . $h_{(k)}(S)$ is a subset that is used as a signature for the set S , and the similarity is compared based on the derived signature by comparing them.

To perform Minhash function efficiently and effectively, a suitable hashing algorithm must be chosen. In the case of hashing documents, the goal is to calculate similarity between sets, therefore the suitable family of hash to use is from the non-cryptographic hash family. Since this research work uses the variant with a single hash function, Murmurhash3 32bit is chosen as the algorithm for Minhash function. Figure 3.8 show the algorithm for Murmurhash3.

3.3.5 Locality Sensitive Hashing

Using Minhash speeds up computation for nearest neighbour search, but it still need to perform comparison to every Minhash representation of the MIST reports in the dataset to find similar malware samples. This will still consume a lot of time as it is an exact method of search using either linear search or space partitioning. A good solution for this is an approximate nearest neighbour search algorithm called Locality Sensitive Hashing. There are a number of variants in the Locality Sensitive Hashing family. For this research work, min-wise independent permutations variant, or Minhash LSH by Leskovec, Rajaraman, and Ullman (2014) is chosen, due to the fact that Minhash is used. Another reason is because the data that is dealt with is text in nature, therefore it is most suitable to be using this variant. Jaccard similarity coefficient in equation 3.1 (page 47) which is used as a similarity measure for this variant as it utilises Minhash, is good for text comparison, as proven in the paper on Minhash by Broder (1997). While the original author did not utilise LSH in the published work, it has been researched and proven by Leskovec, Rajaraman, and Ullman (2014) to be a good algorithm for text similarity comparison.

Using the Minhash algorithm, which is explained in equation 3.2 (page 48) be calculated. Using Minhash signature matrix, the table is divided into band b and row r , where b is the number of bands while r is the number of rows. Therefore, the length of hash needed is $((b*r) = k)$. The probability the Minhash signatures for these documents agree in any one particular row of the signature matrix is $s = J(A, B)$. To calculate similarity using LSH, the probability rule are as follows;

1. The probability that the Minhash signatures agree in all rows of one particular band is $J(A, B)$, where the equation is a Jaccard similarity equation as shown in equation 1 (page 47).
2. The probability that the Minhash signatures do not agree in at least one row of a particular band is $(1 - J(A, B))^r$, where $1 - J(A, B)$ is the dissimilarity measure of Jaccard coefficient and r is the row in the matrix.
3. The probability that the Minhash signatures do not agree in all rows of any of the bands is $(1 - J(A, B))^r$, where $1 - J(A, B)$ is the dissimilarity measure of Jaccard coefficient, and r is row and b is band in the matrix .
4. The probability that the Minhash signatures agree in all the rows of at least one band, and therefore become a candidate pair is $1 - (1 - J(A, B))^r$.

Regardless of the chosen b and r values, the function described above will form an S-curve. The threshold t is the similarity s at which the probability of a candidate is $1/2$, is a function of b and r . From the S-curve graph, as shown in Figure 3.10, it can be seen that the threshold is roughly where the curve is the steepest. Pairs with similarity higher than the threshold are

likely to become candidates, while pairs below the threshold are unlikely to become candidates.

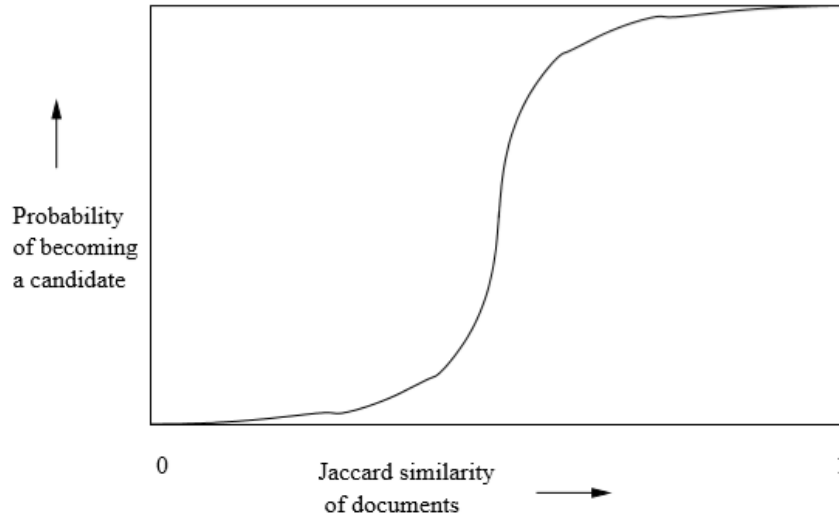


Figure 3.10: LSH S-curve diagram, (Leskovec et al., 2014)

To determine similarity of documents, the similarity threshold must be defined. Let t be the threshold value from $[0,1]$ such that $t = (\frac{1}{b})^{\frac{1}{r}}$. For example, if the signature has length of 400 characters, then $b = 20$ and $r = 20$, and $t = 0.9$, the step value will be $(\frac{1}{20})^{\frac{1}{20}} \approx 0.86$.

Table 3.2: S-curve probability based on $b = 20$ and $r = 20$, $t = 0.9$, and step value ≈ 0.86

J(A,B)	P[A is compared to B]
0.6	0.0007 (0.007%)
0.7	0.0158 (1.58%)
0.86	0.63 (63%)
0.9	0.925 (92.5%)
0.93	0.995 (99.5%)
0.95	0.999861 (99.9%)

Using $t = (\frac{1}{b})^{\frac{1}{r}}$ which is the similarity threshold used in LSH, the values on the right side of Table 3.2 are produced. The Jaccard similarity, $J(A, B)$ values which range from 0 to 1 represents the range, which is shown on the left side of the table. In this study the range that are chosen are 0.6 to 0.95 as it shows an acceptable degree of similarity to compare documents. It can be seen that if the Jaccard similarity as shown on the left side of the table is lower than the threshold which is produced using the $t = (\frac{1}{b})^{\frac{1}{r}}$ equation on the right side of the table, the pair comparison will not be done, but if the Jaccard similarity is greater than 0.9, then the pair comparison will be done. Thus, it proves that LSH can avoid performing all pair comparisons, but still achieve high degree of accuracy.

3.3.6 Clustering

After LSH has performed nearest neighbour search to find similar samples by approximation, clustering is needed to group the similar samples together. Clustering is also called unsupervised learning as it does not require labels for learning, and it is mainly used for learning patterns in the data. In malware clustering case, it can be used for clustering malware families together, as well as discovering unknown malware samples.

3.3.6.1 Disjoint-Set Forest

Based on the study of related works, Disjoint-set Forest algorithm by Tarjan (1975) is chosen as the algorithm of choice for this research work, due to its simplicity, and also speed in clustering. The algorithm has no parameters to adjust and no cluster size need to be predefined before clustering is performed. The algorithm is used to keep a collection of

disjoint sets, by using a representative to represent set members. It has only three operations, which are;

- Make-Set: Create new set with single element
- Find: The step of finding the representative member of the set
- Union: Unification step of two sets

Make-Set is the start of the algorithm where every element is a node by itself, and also a parent pointer to itself. This is indicative of the fact that the element is a representative member of its own set. It has a time complexity of $O(1)$, and the initialisation time is $O(n)$. The pseudocode is shown in Figure 3.11.

```
function MakeSet(x)
    if x is not already present:
        add x to the disjoint-set tree
        x.parent := x
        x.rank   := 0
        x.size   := 1
```

Figure 3.11: Pseudocode for Make Set

The next step of the algorithm is Find. Based on the pseudocode in Figure 3.11, it follows the chain of parent pointer x until the root element is reached. The root element for which x belongs to may be x itself. To make Find algorithm fast, an optimisation method named Path Splitting is used and is shown in Figure 3.12

```

function Find(x)
    while x.parent != x
        x, x.parent := x.parent, x.parent.parent
    return x

```

Figure 3.12: Pseudocode for Find with Path Splitting

After Find is performed, Union is executed. Union uses Find to determine the roots of x and y belong to. The trees are combined if the roots are distinct, and it is done by merging the root of x and y together. At this stage, the algorithm can be very inefficient due to the possibly increasing height of the trees, which grows at a rate of $O(n)$. In order to prevent this from happening, a variant for this algorithm named Union by Rank variant is chosen. The pseudocode for Union by Rank is shown below in Figure 3.13.

```

function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)

    // x and y are already in the same set
    if xRoot == yRoot
        return

    // x and y are not in same set, so we merge them
    if xRoot.rank < yRoot.rank
        xRoot, yRoot := yRoot, xRoot // swap xRoot and yRoot

    // merge yRoot into xRoot
    yRoot.parent := xRoot
    if xRoot.rank == yRoot.rank:
        xRoot.rank := xRoot.rank + 1

```

Figure 3.13: Pseudocode for Union by Rank

The time complexity of this algorithm without any optimisation such as Path splitting and Union by Rank is $O(n)$. By combining the two optimisation methods, the amortised time per

operation is $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function. This function has a value of $\alpha = 4$ for most cases, which makes the operation performing in constant time.

To prove that Disjoint-set Forest algorithm is performing better than other clustering algorithms for which cluster size does not need to be predefined, and for which has been proven to be competitive and used commonly for supervised learning, three other algorithms are also used for comparison purpose, as explained below.

3.3.6.2 Single-linkage Hierarchical Clustering

Single-linkage hierarchical clustering is a variant of Hierarchical clustering created by Sibson (1973). It clusters data in an agglomerative fashion, whereby at each step it combines two clusters together, if the closest pair of elements are not yet part of the cluster. The pseudocode for hierarchical clustering is shown in Figure 3.14 below.

```

1. Set  $\Pi(n + 1)$  to  $n + 1$ ,  $\lambda(n + 1)$  to  $\infty$ 
2. Set  $M(i)$  to  $d(I, n + 1)$  for  $I = 1, \dots, n$ 
3. For  $i$  increasing from 1 to  $n$ 
    if  $\lambda(i) \geq M(i)$ 
        set  $M(\Pi(i))$  to  $\min \{M(\Pi(i)), \lambda(i)\}$ 
        set  $\lambda(i)$  to  $M(i)$ 
        set  $\Pi(i)$  to  $n + 1$ 
    if  $\lambda(i) < M(i)$ 
        set  $M(\Pi(i))$  to  $\min \{M(\Pi(i)), M(i)\}$ 
4. For  $i$  increasing from 1 to  $n$ 
    if  $\lambda(i) \geq \lambda(\Pi(i))$ 
        set  $\Pi(i)$  to  $n + 1$ 

```

Figure 3.14: Single-linkage Hierarchical clustering pseudocode

The advantage of hierarchical clustering is that it creates a hierarchy of data, therefore creating a structure that is more informative than flat clusters. It is also easy to implement, like Disjoint-set forest algorithm. Besides that, initial cluster size does not need to be determined, making it an easy algorithm to use. There are also a number of disadvantages to it, such as time complexity, making it unsuitable for large datasets. It is also very sensitive to outliers. Besides that, the order of the data has a significant impact on the final results. It is also not possible to move instances to other clusters once it has been assigned to a cluster, thus creating room for inaccuracy.

3.3.6.3 DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN) is a clustering algorithm which is based on density of data points (Ester, Kriegel, Sander & Xu, 1996). Given a set of points in some space, if the points are nearby within a fixed radius, it groups together points that are closely packed together. Points which are outside the fixed radius and which are alone in low-density regions are regarded as outliers. Figure 3.15 shows DBSCAN pseudocode.

```

DBSCAN(DB, distFunc, eps, minPts) {
  C = 0
  for each point P in database DB {
    if label(P) ≠ undefined then continue
    Neighbors N = RangeQuery(DB, distFunc, P, eps)
    if |N| < minPts then {
      label(P) = Noise
      continue
    }
    C = C + 1
    label(P) = C
    Seed set S = N \ {P}
    for each point Q in S {
      if label(Q) = Noise then label(Q) = C
      if label(Q) ≠ undefined then continue
      label(Q) = C
      Neighbors N = RangeQuery(DB, distFunc, Q, eps)
      if |N| ≥ minPts then {
        S = S ∪ N
      }
    }
  }
}

```

Figure 3.15: DBSCAN pseudocode

There are some advantages to DBSCAN, such as the cluster size does not need to be determined before clustering is performed. It can also find arbitrarily shaped clusters, and is robust to noise and outliers, due to the usage of minPts, which weakens the single-linkage effect. There are some weaknesses to it as well. The quality of the clustering is very dependent on the distance metric use. Besides that, it has the limitation of not being able to cluster data with different densities.

3.3.6.4 HDBSCAN

Campello, Moulavi & Sander (2013) created Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) as an improvement over DBSCAN. Unlike DBSCAN which performs neighbourhood search based on a fixed epsilon value, HDBSCAN

does so over varying epsilon values, and integrates the result to find a clustering that gives the best stability over epsilon. This enhancement allows HDBSCAN more flexibility in clustering data and finding data of various densities. It is also more robust to parameter selection. Figure 3.16 shows HDBSCAN algorithm.

1. Compute the core distance w.r.t. ϵ for all data objects in X .
2. Compute an MST of G_{mpts} , the Mutual Reachability Graph.
3. Extend the MST to obtain MST_{ext} , by adding for each vertex a “self-edge” with the core distance of the corresponding object as weight.
4. Extract the HDBSCAN hierarchy as a dendrogram from MST_{ext} :
 - 4.1 For the root of the tree assign all objects the same label (single “cluster”).
 - 4.2 Iteratively remove all edges from MST_{ext} in decreasing order of weights (in case of ties, edges must be removed simultaneously):
 - 4.2.1 Before each removal, set the dendrogram scale value of the current hierarchical level as the weight of the edge(s) to be removed.
 - 4.2.2 After each removal, assign labels to the connected component(s) that contain(s) the end vertex(-ices) of the removed edge(s), to obtain the next hierarchical level: assign a new cluster label to a component if it still has at least one edge, else assign it a null label (“noise”).

Figure 3.16: HDBSCAN pseudocode

HDBSCAN is considered a very good algorithm for exploratory data analysis. It is an almost fully automatic algorithm as only the minimum cluster size needs to be set. It has all the advantages of DBSCAN, such as being able to cluster data of varying shapes, as well as being robust to noise and outliers. It is also better than DBSCAN at clustering data of different densities, as well as being much faster than DBSCAN at clustering. To add to that, it also overcomes the weakness of hierarchical clustering, whereby it can automatically

determine the cut off level of the resulting dendrogram. However, it too does have its own weakness. It is not suitable for data with very high dimensionality. For data with more than 150 dimensions, it can be very slow and possibly inaccurate.

3.4 Summary

This chapter discussed the methodology and framework in detail. The framework components are also discussed in detail, as each component contribute to the bigger success of the framework. MalSketch is created with accuracy, speed, and scalability in mind. Thus, to prove the correctness, the supposed increase in accuracy, speed and scalability, experiments and analysis on the results will be done.

CHAPTER 4

EXPERIMENTS, RESULTS AND ANALYSIS

4.1 Introduction

The objective of this chapter is to design and execute experiments for the purpose of evaluating the framework. Analysis on the results will also be done. The experiments are designed specifically for each critical component in the framework, such as Minhash, LSH, and clustering.

For all the experiments conducted, the specifications for all computers used in this research are detailed below.

- **Computer Model:** Dell Precision T3600
- **Processor:** Intel Xeon E5 3.60GHz
- **RAM:** 16 GB
- **Hard Disk:** 1 Terabyte
- **GPU:** Nvidia Quadro K600
- **Host OS:** Ubuntu 16.04
- **Guest OS:** Windows 7
- **OS Virtualisation Software:** Oracle Virtualbox

4.2 Experiment on Minhash Signature

For this experiment, the objective is to find the length of Minhash signature that is suitable for the data that is going to be represented. As it is an approximation of the original representation, some loss of accuracy is to be expected. The goal is to find the suitable length that minimises the lost against percentage of absolute error in Jaccard estimation and running time. This experiment is based on the usage of Minhash with single hash function with permutations, for which Murmurhash3 is used. The dataset which is used for this experiment is a collection of live malware samples from VirusShare (VirusShare, 2017). For this experiment, random sampling will be done to choose 6,500 malware samples out of a total of 65,000 malware samples for Minhash signature length test. Table 4.1 explains the experiment criteria.

Table 4.1: Minhash evaluation parameters

Parameters	Aim
Minhash number of permutations	To evaluate Minhash absolute error in Jaccard estimation
	To evaluate Minhash performance in running time

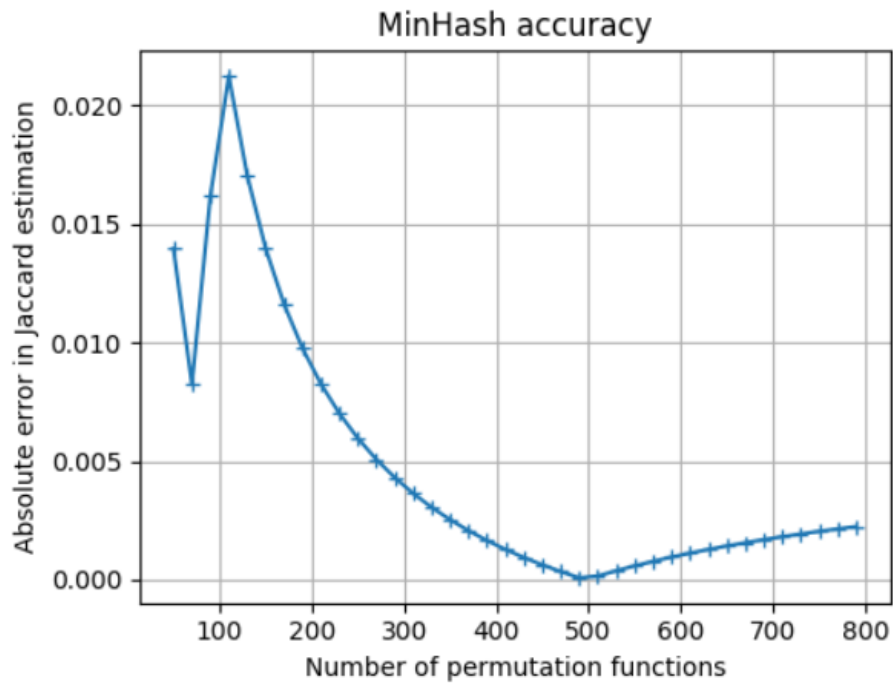


Figure 4.1: Accuracy of Minhash signature based on number of permutations

Figure 4.1 shows a graph of Minhash accuracy based on the number of permutations relative to the absolute error in Jaccard estimation. It is evident from the graph that the longer the signature, the more accurate the Minhash representation of the original data is. From 100 permutations onwards until around 491 permutations, the accuracy is increasing steeply. This also means that the absolute error is decreasing steeply. However, after the length of 491, the relative errors increase steadily, therefore for this research data, Minhash with permutations of 491 is chosen for its very low absolute error.

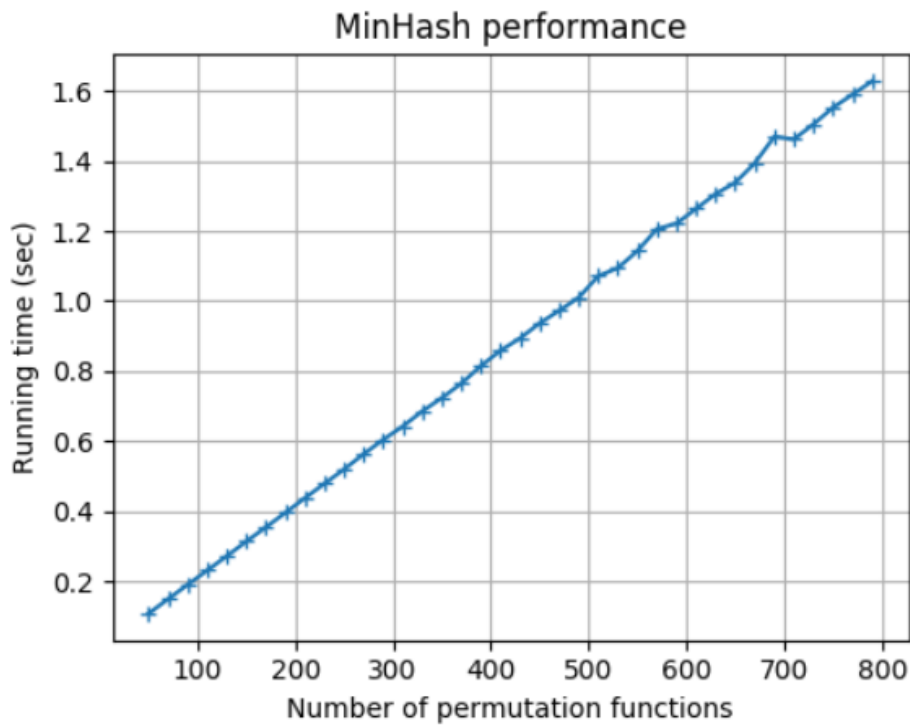


Figure 4.2: Performance of Minhash in terms of running time relative to number of permutation functions

Figure 4.2 shows a graph of number of permutations against running time. As expected, the more permutations are used, the longer the running time. For any cases of using Minhash, there has to be a trade-off of number of permutation functions vs running time, in relation to the absolute error rate. In this research, as the number of permutations chosen is 491, the running time would be around 0.8 seconds per sample. According to both experiments, it gives the lowest absolute error rate. Although the running time is 86 minutes for 6,500 samples, by parallelising the Minhashing process to 8 parallel processes, a time of 9 minutes 24 seconds 537 milliseconds is achieved.

4.3 Experiment on Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) algorithm is chosen as the nearest neighbour search algorithm for this framework due to the efficiency in searching for nearest neighbours, as exact nearest neighbour search is very expensive in terms of time and resources. It achieves the speed by reducing a high dimensional feature space while still retaining a random permutation of relevant features. Research has shown can be used between data sets to determine an accurate approximation of Jaccard similarity (Leskovec, Rajaraman & Ullman, 2014).

For this experiment, which it is based on Minhash LSH, the focus is on the evaluation of the true positive rate, true negative rate, and query time in milliseconds. As it is an approximation nearest neighbour search algorithm, some degree of errors will exist. The error rate is guaranteed, as shown in Chapter 3, subsection 3.3.5. Therefore, an experiment needs to be done to know the accuracy of it. The 3,131 malware samples used for this experiment is from Malheur dataset webpage (Malheur Dataset, 2017). Table 4.2 lists the experiment criteria.

Table 4.2: Experiment criteria for determining true positive and true negative rate, and query time using Malheur dataset

Parameter	Aim
True positive rate	To measure the true positive rate of Minhash LSH
True negative rate	To measure the true negative rate of Minhash LSH
Query time in milliseconds	To measure the Nearest Neighbour Search speed of LSH

It can be seen from the result in Table 4.3 that the accuracy is very high, given that it is an approximation method of nearest neighbour search. The rate of true positive which is at the rate of 97.3%. It is a very high rate for an approximate technique to have. Although it misclassifies the samples into the wrong neighbourhood, it is not something very severe, as corrections can be done to improve the mistakes by using algorithms, or human in the loop. Using algorithms to reassign samples into the correct clusters, and human in the loop will not be explored in this research work. Meanwhile, the rate of true negative which is at the rate of 99.3% is extremely high. A very high true negative rate is a desired trait as it is something that cannot be fixed once the nearest neighbour search or clustering is performed. It is very important especially in cybersecurity field, to have an extremely high rate for true negative as it will be very dangerous to flag a malware as a benign program. The most impressive fact about LSH is the query time, which takes just 1 second 306 milliseconds.

To showcase the speed advantage of LSH, it will be experimented and tested against exact nearest neighbour search algorithm, namely KD-Tree and Balltree. The malware samples used will be from Malheur dataset (Malheur Dataset, 2017). Table 4.4 explains the experiment criteria.

Table 4.3: Results of LSH experiment on Malheur dataset

Parameter	Result
True positive rate	0.973 (97.3%)
True negative rate	0.993 (99.3%)
Query time	1 second 306 milliseconds

Table 4.4: Experiment criteria to measure nearest neighbour search speed

Parameter	Aim
Query time	To measure the Nearest Neighbour Search speed of KD-Tree, Balltree, and LSH

Based on the results shown in Table 4.5 on the query time of KD-Tree and Balltree, it can be said without a doubt that LSH is indeed exponentially faster than the two popular algorithms for nearest neighbour search, and it can still output comparable results as compare to exact nearest neighbour search algorithms.

Table 4.5: Results of nearest neighbour search experiment

Parameter	Result
Query time KD-Tree	2 seconds 682 milliseconds
Query time Balltree	4 seconds 676 milliseconds
Query time LSH	556 milliseconds

4.4 Experiment on Clustering

For evaluating clustering performance and correctness, there are a number of challenges. One of the main challenges is the method of evaluating the correctness. As clustering does not rely on labels, it can be very hard to determine the correctness. Another reason is that some clustering algorithm does not form cluster patterns which are spherical, but form random patterns, making it hard for a lot of evaluation algorithms to measure the correctness of the resulting clusters accurately. However, Silhouette Coefficient algorithm developed by Rousseeuw (1987) does well in determining cluster performance and correctness. It is chosen

for this experiment evaluation, because it is a good clustering evaluation method to evaluate unsupervised learning algorithm, and for unlabelled data.

Silhouette Coefficient is a method of interpreting cluster quality based on the consistency of the data within clusters. It does so by measuring the similarity of an object to its own cluster (cohesion) as compared to other clusters (separation). The score for Silhouette Coefficient lies in the range of -1 to +1, where the higher the value, the more the object is well matched to its own cluster and poorly matched to neighbouring clusters. The value of 0 indicates overlapping clusters. Comparisons will need to be done on a number of clustering algorithms to select the best one for this study, therefore a number of clustering algorithms will be used in the experiment. The algorithms that are included in this experiment include Disjoint-set Forest, Single-linkage Hierarchical clustering, DBSCAN, and HDBSCAN. All algorithms used in this experiment uses Jaccard Coefficient to determine similarities. The dataset that will be used for this experiment is taken from VirusShare (VirusShare, 2017). Malheur dataset which contains 3131 samples will be used in this experiment. Table 4.6 explains the experiment criteria.

Table 4.6: Experiment criteria for cluster quality

Parameters	Aim
Cluster quality	To measure the correctness of the clusters produced by Single-linkage Hierarchical clustering, DBSCAN, HDBSCAN, and Disjoint-set Forest, where the score is in the range of -1 to +1

Based on the results shown in Table 4.7, experiments using Silhouette Coefficient to measure cluster quality has shown that Single-linkage clustering performs the poorest with the score of 0.413. This is probably due to the inherent weakness in Single-linkage Hierarchical Clustering due to the fact that it may merge clusters wrongly, or cut-off the dendrogram at the wrong level, leading to inaccuracy in clustering results. DBSCAN on the other hand has a score of 0.437, which is not far off from Single-linkage Hierarchical Clustering. This could be because it is not flexible in determining clusters of different densities.

HDBSCAN produces a score of 0.544, which is good as it means that the clusters produced are quite coherent. The result can be explained using the nature of the algorithm itself, where it is flexible in determining clusters of different densities and sizes. HDBSCAN also has outlier detection, and it will reject elements which does not belong to any clusters and treated as noise instead. This leads to higher cluster purity. However, with the score of 0.857, Disjoint-set Forest produces the best results out of all the four algorithms used in this experiment. With the perfect score for Silhouette Coefficient being +1, it is quite close to the maximum score. This could be explained with several points; The algorithm used in this experiment has a pairwise checking to reject elements which have similarities below the given threshold, thus resulting in higher cluster purity. Secondly, the clusters which are formed from clustering the results of LSH are somewhat purer, due to the nature of LSH being very strict with threshold boundaries. Thus, it is proven that Disjoint-set forest performs better than Single-linkage Hierarchical Clustering, DBSCAN, and HDBSCAN.

Table 4.7: Results of cluster quality experiment

Parameters	Results
Cluster quality of Single-linkage Hierarchical Clustering	0.413
Cluster quality of DBSCAN	0.437
Cluster quality of HSBSCAN	0.544
Cluster quality of Disjoint-set Forest	0.857

4.5 Experiment of the MalSketch against Malheur

This framework will be tested with Malheur framework which was developed by Rieck et al. (2011). It will be a complete experiment from pre-processing stage to the clustering stage. The dataset used is the same dataset as used by them in Malheur Dataset, (2017) and also the VirusShare dataset (VirusShare, 2017). This is used for comparison to evaluate the accuracy and speed of this MalSketch against Malheur. The reason to compare is because Malheur is a framework widely used clustering malware behaviours, and also due to the fact that it is the most common behaviour analysis framework compared against. Table 4.8 explains the experiment criteria. It serves as a baseline measurement for the MalSketch results.

Table 4.8: Experiment to determine the true positive and true negative rates, and clustering time of Malheur and MalSketch

Parameters	Aim
Accuracy in terms of true positive and true negative rate	To compare the accuracy of Malheur against MalSketch using Malheur and VirusShare dataset
Clustering time	To compare the clustering time of Malheur against MalSketch using Malheur and VirusShare dataset

The experiment is run with the threshold of both frameworks set at 0.75. Results of Malheur as compared to MalSketch is shown below in Table 4.9. From the experiment conducted with 3,131 curated samples from Malheur dataset, the results show that MalSketch has a very high true positive rate, which is at 97.3%, as compared to 89.4% from Malheur. It also has an extremely high true negative rate, which is at 99.4%, as compared to Malheur's 94.2%. Low true positive rates indicate mis-clustering of many samples into the wrong cluster, which is bad as the cluster of families are contaminated. Low true negative rates indicates that the malware family spread across clusters formed are poor. In terms of clustering time, MalSketch is again faster, achieving 1/6 of the time required by Malheur, despite Malheur using prototype clustering method to speed up computation of clusters.

From the experiment with 65,000 non-curated samples from VirusShare dataset, the results show that Malheur takes 5 hours 9 seconds 313 milliseconds to cluster all the samples, despite it using centroid-based Single-linkage hierarchical clustering, which is supposed to speed up the computation by a big factor, as compared to using the regular Single-linkage Hierarchical clustering algorithm. True positive rate is high for this dataset when using Malheur which is at 95%, but true negative rate is low at 85%. The results could be due to the inversion issue inherent in centroid-based clustering algorithms, whereby joined pair are more similar to the parent cluster than to each other. It could also simply be that the algorithm failed to cluster correctly due to the usage of centroids which reduces exact similarity comparisons. MalSketch on the other hand manage to cluster all the data in 5 minutes 2 seconds 737 milliseconds, which is far faster than the time Malheur took to perform the same task. True positive rate is very high at 97.3%, and the true negative rate is 99.1%, which is

extremely high, even when considering that it uses an approximate nearest neighbour search algorithm.

Table 4.9: Results of true positive and true negative rates, and clustering time of Malheur vs MalSketch

	Malheur		MalSketch	
	Malheur dataset	VirusShare dataset	Malheur dataset	VirusShare dataset
True positive	0.894 (89.4%)	0.950 (95%)	0.973 (97.3%)	0.973 (97.3%)
True negative	0.942 (94.2%)	0.850 (85%)	0.994 (99.4%)	0.991 (99.1%)
Clustering time	20 seconds	5 hours 9 seconds 313 milliseconds	1 second 264 milliseconds	5 minutes 2 seconds 737 milliseconds

4.6 Summary

This chapter presented the experiments for evaluating the proposed framework, MalSketch and the analysis of the experiment results. From the experiments performed, it shows that Minhash is indeed a good approximate hash representation of files, as proven by the experiment showing the absolute Jaccard error. It has also been proven that the clustering algorithm of choice is also not only faster, but also outputs better clustering results, as shown by the Silhouette Coefficient test, as compared to single-linkage hierarchical clustering, DBSCAN, and HDBSCAN. For the experiment which compares MalSketch against other framework, specifically Malheur, which this research is inspired from, it has been proven that this framework is indeed not only faster than the framework it is compared against, but also more accurate.

CHAPTER 5

CONCLUSION AND FUTURE WORKS

5.1 Summary

The main objective of this thesis, which was stated in Chapter 1, is to research and design an efficient and scalable automated machine learning-based malware behaviour analysis framework. In Chapter 2, through detailed study of the literatures, some weaknesses of past works have been identified, mainly in the standardisation of malware behaviour reports, the speed of the nearest neighbour search, and also the speed of the clustering algorithm.

In Chapter 3, a detailed explanation of the MalSketch is written. It is based on using a database system to speed up operations, using a standardised behaviour report format suitable for machine learning, approximate nearest neighbour search algorithm for speeding up nearest neighbour search, and a simple but accurate and quick clustering algorithm.

In Chapter 4, experiments are conducted on Minhash algorithm and Locality Sensitive Hashing (LSH) to prove that they are suitable and good replacement for exact nearest neighbour search. Clustering algorithms experiment is also conducted to find a suitable one for use in this framework. Experiment to perform comparison with other framework is also conducted to prove its accuracy and efficiency.

5.2 Future Works

Despite the good results from the framework, many more things are lacking, and can be improved. There are several main improvements which can be done to this framework, which is discussed below.

- **Integration of other malware analysis methodology** – As malware evolves over time, and the difficulty of analysing malware due to the complexity becomes higher, there is a need to evolve the analysis methods as well. Future works should work on integrating static analysis, memory analysis, and image analysis into this framework as well, in order for this framework to be more versatile and also accurate. Besides that, benign program analysis should also be integrated in order to achieve greater accuracy in determining malware samples
- **Minhash** – There are active efforts in the research communities to enhance the accuracy of Minhash and also the speed of computing Minhash. Future works to improve this should work on Minhash algorithms which is more accurate, and also faster in execution speed, as well as having the ability to support streaming.
- **Locality Sensitive Hashing** – While the LSH variant used in this research work is good and efficient, much more can be done to improve its performance and accuracy. By turning the Minhash into binaries, similarities can be calculated in terms of Hamming similarities. This would allow accuracy to increase as the similarity coefficient is able to calculate on a finer grained level.

- **Cluster manipulation** – Future works on this framework should also focus on developing a methodology and use suitable algorithms to evaluate clusters by inspecting elements in the clusters and removing elements when it is deemed not fit for the cluster, and also allow reassignment of elements to other clusters if possible.
- **Distributed computing** – This research work is done in mind of the future. Therefore, most of the algorithms used in this framework has the potential to be used in a distributed computing environment. Future works on this should expand this framework on a distributed computing platform to enhance its performance.
- **Human in the loop** – Human in the loop is not a new term in the AI world. However, in the cybersecurity industry, it is an almost unused terms. By having human analyst to augment the analysis results, greater results can be achieved. Thus, future work on this framework should strive on integrating human in the loop to augment the ability of human analyst to achieve greater potential in doing analysis.

5.3 Conclusion

This research has laid out a solid foundation on malware behaviour analysis. As proven by the experiments, the framework performs extremely well. To ensure that the framework will perform better in the future, continuous improvements must be done to it, and suggestions have also been made on how this framework can continue to be improved upon.

REFERENCES

- Aman, W. (2014). A Framework for Analysis and Comparison of Dynamic Malware Analysis Tools. *International Journal of Network Security & its Applications*, 6(5), 63-74. doi: 10.5121/ijnsa.2014.6505
- Andrade, C., Mello, C., & Duarte, J. (2013). Malware Automatic Analysis. *2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence*. 681-686. doi: 10.1109/brics-cci-cbic.2013.119
- Annachhatre, C., Austin, T., & Stamp, M. (2014). Hidden Markov models for malware classification. *Journal of Computer Virology and Hacking Techniques*, 11(2), 59-73. doi: 10.1007/s11416-014-0215-x
- AVTest. (2019). Malware Statistics & Trend Report. Retrieved from <https://www.av-test.org/en/statistics/malware/>
- Broder, A. (1997). On the resemblance and containment of documents. *Proceedings. Compression and Complexity of Sequences 1997 (Cat. No.97TB100171)*. doi: 10.1109/sequen.1997.666900
- Burnap, P., French, R., Turner, F., & Jones, K. (2018). Malware classification using self organising feature maps and machine activity data. *Computers & Security*, 73, 399-410. doi: 10.1016/j.cose.2017.11.016
- Campello, R., Moulavi, D., & Sander, J. (2013). Density-Based Clustering Based on Hierarchical Density Estimates. *Advances in Knowledge Discovery and Data Mining*, 7819, 160-172. doi: 10.1007/978-3-642-37456-2_14
- CARO. (2018). A New Virus Naming Convention (1991) - CARO - Computer Antivirus Research Organization. Retrieved from <http://www.caro.org/articles/naming.html>

- Cuckoo Sandbox. (2018). Automated Malware Analysis. Retrieved from <https://cuckoosandbox.org>
- Cybercrime will Cost Businesses Over \$2 Trillion by 2019. (2018). Retrieved from <https://www.juniperresearch.com/press/press-releases/cybercrime-cost-businesses-over-2trillion>
- Dahl, G., Stokes, J., Deng, L., & Yu, D. (2013). Large-scale malware classification using random projections and neural networks. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. doi: 10.1109/icassp.2013.6638293
- Dinaburg, A., Royal, P., Sharif, M., & Lee, W. (2008). Ether. *Proceedings of t/he 15Th ACM Conference on Computer and Communications Security - CCS '08*. doi: 10.1145/1455770.1455779
- Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 44(2), 1-42. doi: 10.1145/2089125.2089126
- Ester, M., Kriegel, H., Sander, J., & Xu, X. (1996). A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD'96 Proceedings of the Second International Conference on Knowledge Discovery and Data Mining* (pp. 226-231). Portland, Oregon: AAAI Press ©1996.
- Firdausi, I., lim, C., Erwin, A., & Nugroho, A. (2010). Analysis of Machine Learning Techniques Used in Behavior-Based Malware Detection. *2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies*. doi: 10.1109/act.2010.33

- Gadhiya, S., & Bhavsar, K. (2013). Techniques for Malware Analysis. *International Journal of Advanced Research in Computer Science & Software Engineering (IJARCSSE)*, 3(4), 972-975.
- Galal, H., Mahdy, Y., & Atiea, M. (2015). Behavior-based features model for malware detection. *Journal of Computer Virology and Hacking Techniques*, 12(2), 59-67. doi: 10.1007/s11416-015-0244-0
- Gandotra, E., Bansal, D., & Sofat, S. (2014). Malware analysis and classification: a survey. *Journal of Information Security*, 5, 56-64.
- Gavriliuț, D., Cimpoeșu, M., Anton, D., & Ciortuz, L. (2009). Malware detection using machine learning. *2009 International Multiconference on Computer Science and Information Technology*, 4, 735-741. doi:10.1109/imcsit.2009.5352759
- Hady, M., & Schwenker, F. (2013). Semi-supervised Learning. *Intelligent Systems Reference Library*, 49, 215-239. doi: 10.1007/978-3-642-36657-4_7
- Holland, B., Deering, T., Kothari, S., Mathews, J., & Ranade, N. (2015). Security Toolbox for Detecting Novel and Sophisticated Android Malware. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. doi: 10.1109/icse.2015.235
- Horton, J., & Seberry, J. (1997). Computer Viruses: An Introduction. *Proceedings of the Twentieth Australasian Computer Science Conference – ACSC '97*, 19(1), 122-131.
- Huang, H., Lee, C., Kao, H., Tsai, Y., & Chang, J. (2011). Malware behavioral analysis system: TWMAN. *2011 IEEE Symposium on Intelligent Agent (IA)*. doi: 10.1109/ia.2011.5953604
- Huang, W., & Stokes, J. (2016). MtNet: A Multi-Task Neural Network for Dynamic Malware Classification. *Detection of Intrusions and Malware, and Vulnerability Assessment*, 399-418. doi: 10.1007/978-3-319-40667-1_20

- Islam, R., Tian, R., Batten, L. M., & Versteeg, S. (2013). Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*, 36(2), 646-656.
- Jang, J., Brumley, D., & Venkataraman, S. (2011). BitShred. *Proceedings of the 18Th ACM Conference on Computer and Communications Security - CCS '11*. doi: 10.1145/2046707.2046742
- Jang, J., Woo, J., Yun, J., & Kim, H. (2014). Mal-netminer. *Proceedings of the 23Rd International Conference on World Wide Web - WWW '14 Companion*. doi: 10.1145/2567948.2579364
- Jazdi, N. (2014). Cyber physical systems in the context of Industry 4.0. *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*. doi: 10.1109/aqtr.2014.6857843
- Kaspersky. (2018). Computer Viruses vs Network Worms. Retrieved from <https://www.kaspersky.com/resource-center/threats/viruses-worms>
- Kolosnjaji, B., Zarras, A., Webster, G., & Eckert, C. (2016). Deep Learning for Classification of Malware System Call Sequences. *AI 2016: Advances in Artificial Intelligence*, 137-149. doi: 10.1007/978-3-319-50127-7_11
- Leskovec, J., Rajaraman, A., & Ullman, J. (2014). *Mining of massive datasets*. Cambridge: Cambridge University Press.
- Lin, C. T., Wang, N. J., Xiao, H., & Eckert, C. (2015). Feature Selection and Extraction for Malware Classification. *Journal of Information Science and Engineering*, 31(3), 965-992.
- McAfee. (2018). McAfee Labs Threats Reports – Threat Research. Retrieved from <http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-aug-2015.pdf>

- Mohaisen, A., Alrawi, O., & Mohaisen, M. (2015). AMAL: High-fidelity, behavior-based automated malware analysis and classification. *Computers & Security*, 52, 251-266. doi: 10.1016/j.cose.2015.04.001
- Moon, D., Im, H., Kim, I., & Park, J. (2015). DTB-IDS: an intrusion detection system based on decision tree using behavior analysis for preventing APT attacks. *The Journal of Supercomputing*, 73(7), 2881-2895. doi: 10.1007/s11227-015-1604-8
- Moser, A., Kruegel, C., & Kirda, E. (2007). Limits of Static Analysis for Malware Detection. *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. doi: 10.1109/acsac.2007.21
- Murphy, K. (2013). *Machine learning*. Cambridge, Massachusetts: MIT Press.
- Oprişa, C., Cabău, G., & Sebestyen Pal, G. (2014). Malware clustering using suffix trees. *Journal of Computer Virology and Hacking Techniques*, 12(1), 1-10. doi: 10.1007/s11416-014-0227-6
- Ozsoy, M., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., & Ponomarev, D. (2015). Malware-aware processors: A framework for efficient online malware detection. *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. doi: 10.1109/hpca.2015.7056070
- Pai, S., Troia, F., Visaggio, C., Austin, T., & Stamp, M. (2016). Clustering for malware classification. *Journal of Computer Virology and Hacking Techniques*, 13(2), 95-107. doi: 10.1007/s11416-016-0265-3
- Park, Y., & Reeves, D. (2013). Deriving common malware behavior through graph clustering. *Computers & Security*, 39, 419-430. doi: 10.1016/j.cose.2013.09.006
- Pektaş, A., & Acarman, T. (2018). Malware classification based on API calls and behaviour analysis. *IET Information Security*, 12(2), 107-117. doi: 10.1049/iet-ifs.2017.0430

- Perdisci, R., & U, M. (2012). VAMO. *Proceedings of the 28th Annual Computer Security Applications Conference*. doi: 10.1145/2420950.2420999
- Pircoveanu, R., Hansen, S., Larsen, T., Stevanovic, M., Pedersen, J., & Czech, A. (2015). Analysis of Malware behavior: Type classification using machine learning. *2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (Cybersa)*. doi: 10.1109/cybersa.2015.7166115
- Rieck, K., Trinius, P., Willems, C., & Holz, T. (2011). Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4), 639-668. doi: 10.3233/jcs-2010-0410
- Rousseeuw, P. (1987). Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20, 53-65. doi: 10.1016/0377-0427(87)90125-7
- Sahu, M., Ahirwar, M., & Shukla, P. (2015). Improved Malware Detection Technique Using Ensemble Based Classifier and Graph Theory. *2015 IEEE International Conference on Computational Intelligence & Communication Technology*. doi: 10.1109/cict.2015.147
- Santos, I., Devesa, J., Brezo, F., Nieves, J., & Bringas, P. (2013). OPEM: A Static-Dynamic Approach for Machine-Learning-Based Malware Detection. In A. Herrero et al. (Ed.), *International joint conference CISIS'12-ICEUTE'12-SOCO'12 special sessions. Advances in intelligent systems and computing* (Vol. 189, pp. 271-280). Berlin, DE: Springer.
- Sebastián, M., Rivera, R., Kotzias, P., & Caballero, J. (2016). AVclass: A Tool for Massive Malware Labeling. In F. Monrose, M. Dacier, G. Blanc, J. Garcia-Alfaro (Ed.), *Research in attacks, intrusions, and defenses. RAID 2016. Lecture notes in computer science* (Vol. 9854, pp. 230-253). Cham, CH: Springer.

- Shahzad, F., Shahzad, M., & Farooq, M. (2013). In-execution dynamic malware analysis and detection by mining information in process control blocks of Linux OS. *Information Sciences*, 231, 45-63. doi: 10.1016/j.ins.2011.09.016
- Shalev-Shwartz, S., & Ben-David, S. (2016). *Understanding machine learning*. New York: Cambridge University Press.
- Shijo, P., & Salim, A. (2015). Integrated Static and Dynamic Analysis for Malware Detection. *Procedia Computer Science*, 46, 804-811. doi: 10.1016/j.procs.2015.02.149
- Sibson, R. (1973). SLINK: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1), 30-34. doi: 10.1093/comjnl/16.1.30
- Sihwail, R., Omar, K., & Zainol Ariffin, K. (2018). A Survey on Malware Analysis Techniques: Static, Dynamic, Hybrid and Memory Analysis. *International Journal on Advanced Science, Engineering and Information Technology*, 8(4-2), 1662-1671.
- Symantec Corporation. (2015). Internet Security Threat Report, Volume 21. Retrieved from <https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf>
- Symantec.com. (n.d.). Trojan Horse. Retrieved from: https://www.symantec.com/security_response/glossary/define.jsp?letter=t&word=trojan-horse
- Tarjan, R. (1975). Efficiency of a Good but Not Linear Set Union Algorithm. *Journal of the ACM*, 22(2), 215-225. doi: 10.1145/321879.321884
- Trinius, P., Willems, C., Holz, T., & Rieck, K. (2009). A Malware Instruction Set for Behavior-Based Analysis. Retrieved from <http://www.mlsec.org/malheur/docs/mist-tr.pdf>
- VirusShare. (2017). VirusShare.com - Because Sharing is Caring. Retrieved from <https://virusshare.com/>

- VirusTotal. (2018). Free Online Virus, Malware and URL Scanner. Retrieved from <https://www.virustotal.com/en/>
- Wang, T., Hu, X., Meng, S., & Sailer, R. (2014). Reconciling malware labeling discrepancy via consensus learning. *2014 IEEE 30th International Conference on Data Engineering Workshops*. doi: 10.1109/icdew.2014.6818308
- Wang, P., & Wang, Y. (2015). Malware behavioural detection and vaccine development by using a support vector model classifier. *Journal of Computer and System Sciences*, 81(6), 1012-1026. doi: 10.1016/j.jcss.2014.12.014
- Wang, X., & Yiu, S. (2016). A multi-task learning model for malware classification with useful file access pattern from API call sequence. Retrieved from <https://arxiv.org/pdf/1610.05945.pdf>
- Wuechner, T., Cislak, A., Ochoa, M., & Pretschner, A. (2017). Leveraging Compression-based Graph Mining for Behavior-based Malware Detection. *IEEE Transactions on Dependable and Secure Computing*, 1-1. doi: 10.1109/tdsc.2017.2675881
- Yerima, S., Muttik, I., & Sezer, S. (2015). High accuracy android malware detection using ensemble learning. *IET Information Security*, 9(6), 313-320. doi: 10.1049/iet-ifs.2014.0099
- Zabidi, M., Maarof, M., & Zainal, A. (2011). Ensemble based categorization and adaptive model for malware detection. *2011 7th International Conference on Information Assurance and Security (IAS)*. doi: 10.1109/isias.2011.6122799
- Zaman, M., Siddiqui, T., Amin, M., & Hossain, M. (2015). Malware detection in Android by network traffic analysis. *2015 International Conference on Networking Systems and Security (Nsys)*. doi: 10.1109/nsyss.2015.7043530

- Zhuang, W., Xiao, L., Cui, J., & Chuang, W. (2015). Support Vector Machine Based on Incremental Learning for Malware Detection. In *International Conference on Computer Science and Intelligent Communication (CSIC 2015)* (pp. 204-207). Zhengzhou, China: Atlantis Press.
- Zhou, Y., & Jiang, X. (2012). Dissecting Android Malware: Characterization and Evolution. *2012 IEEE Symposium on Security and Privacy*. doi: 10.1109/sp.2012.16
- Zhu, K., & Yin, B. (2012). Malware Behavior Classification Approach Based on Naive Bayes. *Journal of Convergence Information Technology*, 7(5), 203-210. doi: 10.4156/jcit.vol7.issue5.25

APPENDICES

Appendix A

Web Links

Web link for MIST format.

1. https://github.com/Navein/Cuckoo2MIST/blob/master/cuckoo2mist/conf/cuckoo_elements2mist.xml#L7

Appendix B

Publications

Chanderan, N., & Abdullah, J. (2017). Hierarchical Density-based Clustering of Malware Behaviour. *Journal of Telecommunication, Electronic and Computer Engineering*, 9(2-10), 151-158.

Appendix C

Patents

Title of Invention: Malzeroml: malware behaviour analysis using machine learning

Patent Application Number: PI 2018701796

Filing Date: 04/05/2018