

SCALABLE MALWARE DETECTION USING APPROXIMATE HASH MATCHING

BY

OMAR MOHAMMED OMAR BAWAZEER

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

INFORMATION ASSURANCE AND SECURITY

APRIL 2020

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

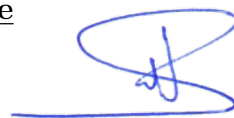
DEANSHIP OF GRADUATE STUDIES

This thesis, written by **OMAR MOHAMMED OMAR BAWAZEER** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN INFORMATION ASSURANCE AND SECURITY**.

Thesis Committee

Sami Zhioua

Dr. Sami M. Zhioua (Adviser)



Dr. Jameleddine Hassine (Member)



Dr. Emad Ramadan (Member)



Dr. Hamoud Aljamaan
Department Chairman

Dr. Salam A. Zummo
Dean of Graduate Studies

23-4-2020

Date



©Omar Mohammed Omar Bawazeer
2020

Dedication

I dedicate this work to the pure spirit of my father, my mother, brothers, sisters, wife, children, and all those who have supported me and prayed for me.

ACKNOWLEDGMENTS

I would like to express my appreciation and gratitude to my supervisor, Dr. Sami

Zhioua, for his unstinted support, guidance, and advice to me.

*I am as well indebted to my committee members for their encouragement and helpful
suggestions and time.*

*A great thanks to the community of the ICS department and the KFUPM for their
effort and allowing me to be one of their society.*

TABLE OF CONTENTS

ACKNOWLEDGMENTS	vi
LIST OF TABLES	x
LIST OF FIGURES	xii
ABSTRACT (ENGLISH)	xiv
ABSTRACT (ARABIC)	xvi
CHAPTER 1 INTRODUCTION	1
1.1 Malware Detection Techniques	4
1.1.1 Signature-Based Techniques	4
1.1.2 Anomaly-Based (Behavior-Based) Techniques	5
1.1.3 Heuristic-Based Techniques	6
1.2 Similarity Hashing	6
1.3 Problem Statement	8
1.4 Thesis Contributions	9
1.5 Thesis Organization	11
CHAPTER 2 LITERATURE REVIEW	13
2.1 Malware Detection Analysis	13
2.1.1 Static Analysis	13
2.1.2 Dynamic Analysis	14
2.1.3 Hybrid Analysis	15

2.2	Malware Detection Using System Events	15
2.3	Hash Based Matching	20
CHAPTER 3 EXPERIMENTAL SETUP AND DATA COLLEC-		
	TION	24
3.1	Experimental Environment: ProPeBox Environment	25
3.1.1	Experiment Automation	27
3.1.2	Converting PML Report Files to CSV	29
3.2	Data Collection	30
3.2.1	Malware Dataset and Collection Procedures	31
3.2.2	Malware Events and API Calls Dataset	33
3.3	Ground Truth Labeling	35
CHAPTER 4 APPROXIMATE HASH BASED MATCHING		
	(AHBM) APPROACH	40
4.1	Approach Motivation for Malware Detection	40
4.2	AHBM Signature Generation	42
4.2.1	Criterion of Features Selection for API Calls of Malware	42
4.2.2	Generating Fingerprints and Similarity Digests	47
4.3	Similarity Comparison of AHBM Digests	50
4.3.1	Selecting The Threshold Value for AHBM	53
CHAPTER 5 MALWARE SIGNATURE GENERATION USING		
	THE LONGEST COMMON SUBSTRING (LCS)	55
5.1	Longest Common Substring (LCS)	55
5.2	Simplified Method of LCS for Extracting Signatures of Malware	56
5.2.1	Implementation of Simplified Method of LCS Algorithm	58
5.2.2	Filtering Process of The LCS Strings	59
5.2.3	Cleaning Process of The LCS Strings	61
5.3	The Criteria for Selecting The Malware Signatures Based on The Sim- plified Method	64

5.4	Thresholds Values for Selecting Signature Criteria	66
5.5	Choosing The Malware Signatures Based on The Simplified Method of LCS	68
CHAPTER 6 MALWARE DETECTION BASED ON API CALLS USING AHBM APPROACH		69
6.1	Malware Detection Model	69
6.2	Malware Detection	72
6.2.1	Cross-Validation Technique	72
6.3	Results and Discussion	76
6.4	Scalability and Time Efficiency	81
CHAPTER 7 CONCLUSION AND FUTURE WORK		85
7.1	Summary of Thesis Contributions	85
7.2	Future Work	88
REFERENCES		89
VITAE		96

LIST OF TABLES

2.1	Comparison Between Static and Dynamic Analysis	14
3.1	Host Machines Specification for The System and Malware Events Collecting Experiments	25
3.2	Time Parameters of Different Stages of ProPeBox Environment	32
3.3	Malware Families of VirusShare Datasest	38
3.4	Malware Families of VirusTotal Datasest	39
4.1	Malware API Calls vs Representative Characters	43
5.1	Specification of Hyper Performance Computing Machine at KFUPM	58
5.2	Time-Consuming for Generating Signatures Using LCS Algorithm Based on Simplified Method	59
6.1	Training and Testing Samples for Each Malware at Each Fold	73
6.2	The Malware and Samples Number of VirusShare and VirusTotal Datasets	74
6.3	Machine Specification for Malware Detection Experiments	77
6.4	Training Results of VirusShare Dataset to Evaluate The Malware Detection Model Using Cross-Validation Technique	77
6.5	Testing Results of VirusShare Dataset to Evaluate The Malware Detection Model Using Cross-Validation Technique	77
6.6	Training Results of VirusTotal Dataset to Evaluate The Malware Detection Model Using Cross-Validation Technique	78

6.7	Testing Results of VirusTotal Dataset to Evaluate The Malware Detection Model Using Cross-Validation Technique	78
6.8	Results of Malware Detection Using Cross-Validation Evaluation Technique	81
6.9	Results of Scalability and Time Efficiency Experiments for VirusShare Dataset	82
6.10	Results of Scalability and Time Efficiency Experiments for VirusTotal Dataset	83
6.11	Time Results of Scalability and Time Efficiency Experiments for VirusShare Dataset	83
6.12	Time Results of Scalability and Time Efficiency Experiments for VirusTotal Dataset	83

LIST OF FIGURES

1.1	Malware Detection Techniques	4
2.1	Malware Detection Analysis	13
3.1	Structure of ProPeBox Environment	26
3.2	Automation Process for Recording System Events Using ProPeBox En- vironment	28
3.3	Converting PML Report Files to CSV	30
3.4	Families of VirusShare Malware Samples Dataset	36
3.5	Families of VirusTotal Malware Samples Dataset	37
4.1	Data Object and Features (Real API Calls)	42
4.2	Data Object and Features Using Representative Set of Characters . . .	44
4.3	Data Object and Features	45
4.4	Example Explained Popularity Value V_{pop} Calculation Using Sliding Window of Size 5 and Data Sample of Size 18	46
4.5	Process of Inserting Features Inside Bloom Filter	49
5.1	Example Illustrated The Length of API Calls of All Samples of One Malware	57
5.2	Example Illustrated The Findings State of The LCS Strings Based on The Simplification Method	60
5.3	The Results of Filtering Process on The Findings of LCS Strings Based on The Simplification Method and Preparing The Input and Output Lists for Cleaning Process	61

5.4	Cleaning Process Procedures	62
5.5	Statistics of LCS Strings Based on Simplified Method	65
5.6	Thresholds Values for Selecting Signature Criteria	67
6.1	Malware Detection Model	70
6.2	Example of Generating Bloom Filters of Malware Training Samples and Signature	74
6.3	Malware Detection Using AHBM Approach	75
6.4	Malware Results (TP, FN, FP, and TN)	81
6.5	Malware Results (Accuracy, Precision, Recall, and F1)	82

THESIS ABSTRACT

NAME: OMAR MOHAMMED OMAR BAWAZEER

TITLE OF STUDY: SCALABLE MALWARE DETECTION USING APPROXIMATE HASH MATCHING

MAJOR FIELD: INFORMATION ASSURANCE AND SECURITY

DATE OF DEGREE: April 2020

Nowadays, malware detection is a crucial field in the security world. In order to detect malicious activities in host machines, many techniques have been proposed. These techniques can be classified as static, dynamic and hybrid. They are implemented using different approaches and methodologies, such as machine and deep learning, neural networks, and others. The main idea of malware detection systems is to define the conspicuous recognizable features for a given malware and consider them as its signatures. These signatures are then compared with host machines activities logs using similarity computation. The methods, tools, and techniques vary based on their capabilities, speed, scalability, and malware detection rate.

By exploring the digital forensics field, we noticed the existence of several tools used for detecting the similarity amongst data objects. In particular, the Approximate

Hash-Based Matching (AHBM) approach is known to be very scalable and can deal with colossal events log data very efficiently. It has several advantages compared to the existing hash-based matching approaches, namely, using a space-efficient manner to represent the signatures using Bloom filters, efficiently and speedily similarity detection, quick signatures generation, etc. This inspired us to use the AHBM approach for detecting the malicious activities in the host machines.

In this thesis, we study the capability of using the AHBM approach for malware detection based on API calls. For this purpose, we developed a new protected and controlled environment, namely, ProPeBox, to collect malware events traces including their API calls. ProPeBox is a protected environment since it can safely launch the malware and efficiently record their activities with protecting the host machine from malware infection. Given that AHBM is mainly dependent on string matching, we have implemented the Longest Common Substring (LCS) using our simplified method to extract the longest sequences of malware API calls and consider the sequences that meet the signatures selection criteria as the signatures of malware.

The findings of our experiments show promising results for using the AHBM approach in the malware detection field. It also proves that the effectiveness of using our simplified method for extracting the sequences of malware signatures using the LCS algorithm. It also demonstrates the scalability of the AHBM approach when it deals with a massive amount of malware data objects.

ملخص الرسالة

الاسم:

عمر محمد عمر باوزير

عنوان الدراسة:

كشف البرامج الخبيثة بواسطة أحداث النظام باستخدام اداة التشابه الجزئي

التخصص:

أمن و ضمان المعلومات

تاريخ الدرجة العلمية: ابريل 2020

في الوقت الحاضر ، يعد اكتشاف البرامج الضارة مجالاً مهماً في عالم الحوسبة. لتحقيق هذا الهدف ، تعد الأساليب الثابتة والديناميكية والهجينة هي التقنيات الرئيسية التي يتم استخدامها للكشف عن الأنشطة الضارة في أجهزة المستخدمين. يتم تطبيق هذه التقنيات باستخدام العديد من الأدوات والطرق مثل التعلم الآلي والعميق ، والشبكات العصبية ، وغيرها. تتمثل الفكرة الرئيسية لأنظمة الكشف عن البرامج الضارة في تحديد الميزات التي يمكن التعرف عليها بشكل واضح للبرامج الضارة واعتبارها بمثابة توقعات للبرامج الضارة. ثم ، يتم استخدام التوقع لحساب التشابه بينها وبين أنشطة أجهزة المستخدمين. تختلف هذه الأساليب والأدوات والتقنيات وفقاً لقدراتها وسرعتها وقابليتها للتوسع ومعدل اكتشاف البرامج الضارة.

من خلال استكشاف مجال الطب الشرعي الرقمي ، نجد أن هناك العديد ممن خلال استكشاف مجال الطب الشرعي الرقمي ، نجد أن هناك العديد من الأدوات المستخدمة للكشف عن التشابه بين كائنات البيانات. من المعروف أن المطابقة التقريبية المستندة إلى التجزئة قابلة للتطوير ويمكنها التعامل مع بيانات سجلات الأحداث الضخمة بكفاءة عالية. ولها العديد من المزايا مقارنة مع الأساليب الحالية ، وهي استخدام طريقة فعالة من حيث استغلال المساحة لتمثيل التوقع باستخدام فلاتر بلوم وكشف التشابه بكفاءة وسرعة الخ. وقد ألهمنا ذلك لاستخدام نهج التشابه الجزئي للكشف عن الأنشطة الضارة في أجهزة المستخدمين.

في هذه الرسالة ، ندرس إمكانية استخدام نهج التشابه الجزئي للكشف عن البرامج الضارة بناءً على استدعاءات واجهة تطبيق البرنامج. و لهذا الغرض ، قمنا بتطوير بيئة جديدة ومحمية ويمكن التحكم بها ، وهي ProPcBox ، لجمع آثار أحداث البرامج الضارة و استدعاءات واجهة برمجة

التطبيقات الخاصة بهم. تعتبر ProPeBox بيئة محمية لتشغيل المايور بشكل آمن و تسجيل احداثها بكفاءة مع الحفاظ على سلامة النظام المضيف للبيئة من الاصابة أو التأثير بالمايور. نظرًا لأن التشابه الجزئي يعتمد بشكل أساسي على مطابقة السلاسل النصية، قمنا بتنفيذ خوارزمية أطول سلسلة فرعية مشتركة باستخدام طريقتنا المبسطة لاستخراج أطول السلاسل النصية من استدعاءات واجهة برمجة تطبيقات البرامج الضارة. وبالتالي يمكن اعتبار السلاسل النصية و التي تلي معايير اختيار التوقع كتوقعات للبرامج الضارة.

تظهر تجاربنا نتائج واعدة لاستخدام نهج AHBM في مجال الكشف عن البرامج الضارة. و تثبت هذه النتائج أيضًا فعالية استخدام طريقتنا المبسطة لاستخراج تسلسلات توقعات البرامج الضارة باستخدام خوارزمية LCS إلى جانب ذلك ، تكشف النتائج إمكانية استخدام منهج AHBM للكشف عن البرامج الضارة. كما توضح النتائج قابلية التوسع في نهج AHBM عندما يتعامل مع كمية هائلة من كائنات بيانات البرامج الضارة.

CHAPTER 1

INTRODUCTION

Malicious Software (aka malware) is a program, software, or code that is designed to harm, damage, or control digital resources such as computers, mobile devices, and networks. Today, malware attacks use more efficient mechanisms to hide their activities inside the victim's resources to evade detection by malware detection techniques. In order to mitigate the impact of these increasingly sophisticated malware attacks, computer scientists have been working to develop better detection techniques.

Application Programming Interfaces (APIs) is a combination of functions and data structures that allow programs (malware) to interact and ask the operating system to perform and execute specific instructions such as creating a file, deleting a registry key, connecting to a remote server, etc. Windows operating system is calling the APIs to perform its functions. Therefore, the study of API calls is widely used with different techniques, methods, and tools to understand, recognize, and detect malware behavior.

Many antivirus products use signature-based techniques to identify the malicious files and activities by comparing them with the previously known and stored signa-

tures in the antivirus databases. Signature-based techniques are good and fast for recognizing known malware. But they fail in detecting new malware whose signatures have not yet been stored in the databases of antivirus applications. To address such deficiency, several security applications tend to use anomaly-based (behavior-based) techniques along with signature-based techniques to improve detection of known and unknown malware. Additionally, researchers have adopted more sophisticated methods, such as using artificial intelligence to develop malware detection operations. They have introduced various classification algorithms to get a higher level of detection of new and unknown malware.

In recent years, Approximate Hash-Based Matching (AHBM) approach has emerged and has been used in several computer industries and forensics researches. Because AHBM is not integrated with the existing digital investigation tools, and knowledge of the technique is still limited, AHBM has not gained mainstream popularity. In brief, AHBM is used to identify similar versions of files and fragments recovered from deleted files or memory [1].

Cryptographic hash functions or block-based hashing are the most famous generation of hashing that were designed to find similarities (integrity) between different binary data [2]. Cryptographic hash functions are used for many security purposes and applications such as digital signatures, authentication, Message Authentication Code (MAC), one-way password file, virus detection, intrusion detection, to construct a pseudorandom function (PRF), pseudorandom number generator (PRNG). It is efficient and fast, but the similarity result will be strongly affected if there is a trivial change on a bit in one of the compared files.

Introduced in 2006 by Dr. Kornblum, SSDEEP is even more robust than block-based hashing. The idea of SSDEEP relies on breaking up the input file into several chunks through a rolling hash function. Then, different hash functions will be used for hashing all chunks and producing a tiny digest for each chunk. The signature will be built depending on the concatenation of hashes for all of the individual chunks [2], [3], [4].

The most powerful and accurate similarity hashing tool appeared in 2010 when Roussev introduced Similarity Digest Hash (SDHASH) [5]. SDHASH is a practical hash-based matching tool to identify, represent, and match chunks of data [1]. It extracts statistically improbable features, hashes them, and stores them in the bloom filters [4]. Unlike digests of block-based hashing that only give yes or no answers because of their similarity query, SDHASH generates a range between 0 to 100 as a similarity score. So, we can tell at what extent the compared data are correlated [6].

AHBM provides the right solution for similarity detection in terms of efficiency and performance, so this motivates us to use it as a technique for malware detection to try to reduce the time-consuming detection process and increase the performance and scalability of the detection process. Also, it preserves the privacy of user data during the detection process because the AHBM will generate the hashes of the data and use it through other detection phases instead of the data itself.

1.1 Malware Detection Techniques

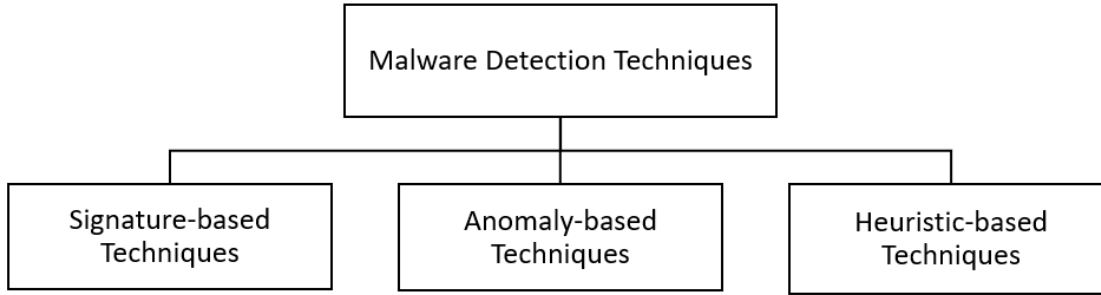


Figure 1.1: Malware Detection Techniques

1.1.1 Signature-Based Techniques

A signature is a unique sequence of bits that identifies a specific family of malware [7] [8]. Most antivirus applications initially used signature-based techniques to detect various types of malware by matching them with their known stored signatures [7], [9], [10], [11].

The signatures have been stored in the databases of antivirus applications to be used during the detection process [7] [12]. Antivirus applications look for a pattern of bits by disassembling the infected file code [7]. Then, the process of features extraction extracts useful features to construct the signature for a particular malware family [11]. The advantages of this type of detection are simplicity, efficiency, and relatively faster than other techniques [9], [10], [12]. Signature-based techniques have small error rate [9]. However, this type of detection technique is vulnerable to certain malicious attacks, such as zero-day attacks [8]. Also, it cannot detect newer malware signatures. The databases of signatures need to be updated regularly[10] and require a relatively

long time to produce a unique signature [9].

1.1.2 Anomaly-Based (Behavior-Based) Techniques

Anomaly-based detection techniques are used to monitor system behaviors to identify abnormal and anomalous activities. Anomaly-based techniques have two main stages: the training stage (learning stage) and then the detection stage (monitoring stage). The goal of the learning phase is to train the detector the normal system behavior. After that, in the monitoring phase, the detector can differentiate between normal and abnormal behavior and detect the anomalous activities of the system [8] [9] [10] [11] [12]. The primary advantage of using anomaly-based or behavior-based techniques is the ability to identify and detect unknown malware that signature-based detection cannot detect. However, the high false-positive rate and difficulty of choosing the appropriate features for the training phase are the basic disadvantages of the anomaly-based techniques [9] [13].

Specification-based malware detection techniques are a special type of anomaly-based techniques. Specification-based techniques try to approximate the system requirements instead of the system implementations [8]. In the specification-based techniques, the learning stage is the attainment of several rulesets, which identifies all the legitimate behavior any application can show for the system being monitored [13]. This type tends to reduce the false positive rate and to increase the false-negative rate [7].

1.1.3 Heuristic-Based Techniques

Some researchers in their papers like [9] and [12] mentioned the heuristic-based malware detection techniques as independent techniques besides signature-based and anomaly-based techniques. Whereas, the authors of [7] joined them with behavior detection techniques. In general, Heuristic-based detection techniques came to overcome the disadvantages of other techniques [9]. These techniques use artificial intelligence techniques such as machine learning, data mining [9], neural networks, and fuzzy logic [12] for training and learning system behaviors. Heuristic-based techniques applied in two steps: firstly, the system observations will be recorded without any threatened or attacks to study system behavior. Then, the collected information from the first step is used to check and detect in case of threats or attacks [7]. However, the main limitation of those techniques is the high false rate.

1.2 Similarity Hashing

Generally, approximate matching can be classified into three types: content, perceptual, and hash-based matching. The algorithms of content and hash-based matching are designed to find the similarity between binary data such as executables, network traffic, memory dumps, and documents. Whereas, the perceptual algorithms are designed to match videos and pictures [1].

Two methods we can use to judge the similarity between two files are syntactic and semantic. From the perspective of the computer, the similarity is called syntactic similarity, and from a human standpoint, the similarity is called semantic similarity.

If two files are containing the same data, the files are semantically identical. Documents that can be semantically similar may not be represented in similar formats on the hard disk, but when expressed a human can recognize the similarities. For example, a Microsoft Word and a PDF document are semantically identical if they communicate the same pages. And when we have two files (let's say "A" and "B") and the hashes of them are identical ($h(A) == h(B)$), then A and B are syntactically identical. Documents that can be syntactically similar will be represented in similar formats on the hard disk [1].

Block-based hashing is the most famous hashing that can be used to identify the binary data similarity. It is fast and computationally efficient. Many of existing hash functions have supported block-based hashing. On the other hand, the main drawback of similarity hashing using block-based hashing is related to the pre-image resistance property of hash functions. For instance, if you have two similar files with minor differences at the beginning, the hash digests of both of them will be different, and the similarity result will generally lose [2].

Syntactical similarity can be measured by hash-based matching between files through estimating data commonalities on the byte level. The advantage of using byte-level similarity is to allow the hash functions to find the similarity between unknown content types so, the approximate hashing of unstructured and complex data like network packets, memory images, and documents will be allowed [1].

Currently, according to the community of digital forensics, the most considerable hash-based similarity tools that acquired more scrutiny than others are SDHASH and SSDEEP. Both search for chunks of statistically improbable bytes known as features.

Features are used to compute the similarity hashing between files [1].

SSDEEP is the main example of context-triggered piecewise hashing. The SS-DEEP algorithm divides the file content into several chunks after defining a context of data within an object of binary data. And the hash result is the concatenation of hashes for all of the individual chunks. But the primary issue in context-triggered piecewise hashing is the fact that the hash that is composed by concatenating the hash of all chunks. So, the order of chunks is critical. If the insertion or deletion is done on one of the compared files, the similarity result will be affected [2].

In 2010, SDHASH was introduced as empirical hash-based matching to identify, represent, and match chunks of data [1]. It was applied a statistical method to select features of fingerprinting. SDHASH aimed to reduce the false positive rate by choosing the most likely unique features of data object [6]. It can be implemented as the Approximate Matching tool in three scenarios: the contraband presence detection, internal data detection for unauthorized copies, and data detection on unauthorized exfiltration [14].

1.3 Problem Statement

Detecting malware events in log datasets (Network traffic, executable binaries, system event logs) can be done using machine learning and data mining techniques. While such methods might be very accurate, the main problem remains, their scalability. Indeed, for massive datasets, machine learning techniques are not scalable enough. Large datasets are very common in practice, for example, in case a network security

administrator is receiving event logs of several machines. Consequently, detecting malware signatures can be very difficult because it requires matching all malware signatures with all system logs. Sometimes, a small shift in the log entries will result in detection failure.

At the same time, malware is using continuously changing behavior, which requires regular checking of the system event logs. So, in digital forensics and security, the need for fast, efficient, and scalable tools is increasing to help the investigators sift through large datasets. The idea we're proposing is to use techniques from digital forensics to identify malware API calls efficiently. Digital forensics techniques are known to be very scalable and can deal with substantial log data very efficiently. Most importantly, it can tell if a given malware signature can be found in a large number of event logs very quickly and without splitting the input data as blocks.

Besides, using a new approach in the malware detection field requires collecting a dataset containing all possible details of malware events. So, this step arises the need for obtaining a new dataset for serving this research. The collected dataset can be used for training and testing the techniques and properly extracting the signatures of malware to gain accurate results.

1.4 Thesis Contributions

One of the main goals of this research is to investigate existing hash-based matching techniques used typically for digital forensics and identify the most applicable one for malware detection in large system event logs. Hence, we found that the Approximate

Hash Based Matching (AHBM) approach has several advantages amongst other methods. Moreover, to the best of our knowledge, no existing malware detection techniques based on malware API calls used (AHBM). So, in this research, the AHBM approach has been presented as a new approach used for malware detection.

ProPeBox, a new protected and controlled environment, is developed and introduced for collecting and recording the malware events. ProPeBox is a protected environment since it can safely launch the malware and efficiently record their activities with protecting the host machine from malware infection. It has produced rich malicious datasets for multi-purposes in research. As a result of using ProPeBox, VirusShare, and VirusTotal datasets are collected and utilized in our study.

Furthermore, the simplified method of the Longest Common Substring (LCS) algorithm is presented in this research to extract the malware signatures from the sequences of malware API calls. Indeed, we have exploited the working nature of AHBM to optimize the execution time of the LCS algorithm to generate the signatures.

The most important contribution in this research is to use the AHBM approach for malware detection and provide empirical evidence about the effectiveness and scalability of the AHBM approach in detecting malware events. We have implemented the cross-validation technique to evaluate the malware detection model, and we have got promising results that open several research ideas to improve and develop this approach for malware detection.

1.5 Thesis Organization

This first chapter is an overview of the malware detection techniques and hash techniques, including similarity hashing. Besides, the problem statement and the thesis contribution have been shown.

The second chapter is the literature review chapter. We have summarized the types of malware detection analysis and expressed the literature of malware detection using API calls. The research of hash-based matching is shown as well.

In the third chapter, the procedures of data collection are presented, and the newly developed environment, namely, ProPeBox, is introduced. Besides, the malware dataset is shown, and the ground truth labeling steps are explained.

Approximate Hash Based Matching is presented in the fourth chapter. Firstly, we explain the approach motivation and then give a technical discussion about how we can use the AHBM for malware detection. Several sections, including AHBM signature generation, criteria of feature selection, digest similarity comparison, and selecting the threshold value for AHBM, are introduced.

In the fifth chapter, we discuss and explain the procedures of conducting our experiments, including extracting the malware signatures by using the simplified method of the Longest Common Substring (LCS) algorithm, filtering and cleaning the findings of the simplified method, and discussing the criteria of selecting the malware signatures.

After that, the malware detection model is expressed in the sixth chapter. We have discussed all stages of the malware detection model. Then, we evaluate our model using a cross-validation technique. As well as we discuss the malware detection

model results and the scalability and time efficiency of the model.

In the last chapter, we summarize our contribution and thesis conclusion along with future work. And the limitation of our research is discussed.

CHAPTER 2

LITERATURE REVIEW

In this chapter, we summarize the types of malware detection analysis and give an overview of the literature of malware detection using API calls. The research of hash-based matching is shown as well.

2.1 Malware Detection Analysis

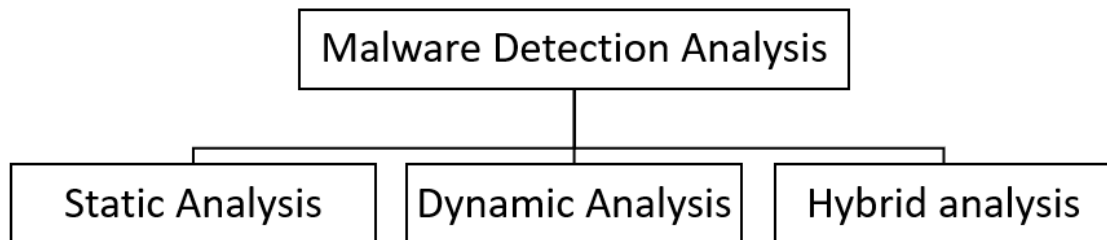


Figure 2.1: Malware Detection Analysis

2.1.1 Static Analysis

Static analysis can be implemented by analyzing malicious files without executing them to extract information from malware executables or their binary code. The

purpose of this process is to understand how malware functions are performed and what system APIs use so that developers of antiviruses and security applications can protect computer systems against them. To achieve that, the malware code will be reversed engineered, and the malware code structure will be analyzed. several tools are used for static analysis such as decompiler, debugger, source code analyzer, and disassembler [7][10][11].

2.1.2 Dynamic Analysis

Dynamic or behavior analysis can be implemented on malware by executing them in a protected environment and monitoring the malware behavior in the host machine and how the system interacts with them as well as analyzing the infected files, system APIs, calls, events, and control flow. Many tools are used for building protected testing environments like a sandbox, virtual machine emulator, simulator [7][10][11].

Table 2.1: Comparison Between Static and Dynamic Analysis

Comparison	Static Analysis	Dynamic Analysis
Time	safe and fast	consuming more time
Detection	only known malware	known and unknown malware
False positive rate	low false positive rate	high false positive rate

One of the known dynamic analysis techniques is using system calls and events. Many types of research have been conducting for malware detection, classifying, and clustering using API calls. Detection systems can use API calls through two main methods, which are calculating API frequency and extracting the API sequence. An API frequency-based technique can be done by selecting the vital features amongst all malware API calls and counting them to define the detection system threshold.

Whereas, other detection systems have been identifying the distinct API sequences to recognize the malicious activity in the computer system. Both methods can be implemented with static or dynamic malware analysis techniques. These methods, generally, are used by machine learning techniques to build a malware detection, classification, or clustering system.

2.1.3 Hybrid Analysis

In a hybrid analysis, static analysis and dynamic analysis are combined to take the advantages of both. So, the malware signatures will first be gathered by analyzing the malware code, and then their behavior will be monitored inside an isolated environment [7][10][11]

2.2 Malware Detection Using System Events

Malware detection and classification systems can be divided into two main directions in terms of how to use system events as detection features. The first one has selected an assortment of vigorous events and calculate the frequencies of them to determine the threshold that differentiates between benign and malicious activities. And the other direction has used a distinct sequence of system events and API calls to reveal malware activities. Static and dynamic analyses are individually, or both implemented for selecting the features of malware detection systems. Generally, Machine learning and deep learning algorithms are widely used for malware classification and detection systems.

Research works [15], and [16] developed a system called intelligent malware detection system that contained three main stages: API execution sequences parser, object-oriented association rule generator, and rule-based classifier. They created the PE file signatures by using the API calls. Those signatures are contained in six fields ((ID and name of PE file), type (benign or malicious), and (ID, name, and the total number of called API sequences)). This system achieved a 93.07% accuracy, which got a better result than other data mining techniques such as SVM, J4.8, and Naive Bayes.

Masud et al. introduced a model called hybrid feature retrieval (HFR) to detect malware. This model extracted three types of features at various levels of abstraction from executables and then integrated them to contain one set of features (hybrid features set). Those features are binary n-gram, derived assembly, and DLL call features that are used by SVM and J48 classifiers to detect malware executables. Their model achieved around 97% and 96% on accuracy for 1st and 2nd datasets, respectively [17].

Flaglien et al. introduced an identification technique for malware based on forensic data from several machines on the network, and then they found the correlation between forensic evidence. They used many features of files from complex and large dataset such as file metadata (timestamp, file name, type, allocation status, file system entry,...), case metadata (file ID, machine ID, media ID), and file content-based data (file content type, MD5 hash value, file entropy value,...) to identify and correlate malware infection between 5 machines in the network [18].

Bai et al. used data mining algorithms to build a scheme for malware detection.

They extracted 197 features from portable executables to distinguish between benign and malware software, then they implemented CfsSubsetEval and WrapperSubsetEval to select the effective subset of features from the 197 features. After that, they classified them by j48 and random forest learning and got 99.1% on accuracy and 97.6% for identifying new malware [19].

Fan et al. papered their system to detect malware using data mining techniques based on API logs. To reduce the complexity and increase the accuracy, they monitored only six DLLs (user32, kernel32, advapi, ntdll, ws2_32, and wininet). This provided many valuable functionalities, like easier network access and improved process management. They collected their dataset by using a virtual machine in two stages. In the first stage, they detected the behaviors of code injection using their TraceHook tool. Then, they restarted the VM and monitored the API calls of each sample. They focused on the sample process itself with their children and injected processes. Finally, they classified the collected data using Support Vector Machine (SVM), Decision Tree (J48), and Naive Bayesian. They got the best result on the accuracy of 95% with J48 using only 80 attributes [20].

The authors of [21] used neural network technique to classify malware call sequences inside the system. And to get the best features for classification, they constructed their neural network technique based on the recurrent network layer and convolutional. The researchers recorded the kernel APIs that were called during the malware sample execution and used them for malware characterization. Besides, they made a boolean vector for each malware that attached the absence or presence of VirusTotal antivirus signature. Then, Kolosnjaji et al. implemented the DBSCAN

algorithm and cosine distance to that signatures. This method achieved 85.6% and 89.4% on precision and recall, respectively.

The researchers in [22] used an emulator and VMware to record malware system calls in both environments and use that calls to detect split personalities through recognizing the changes in the behavior of the same malware.

Calculating frequencies of API functions is a common method in malware detection systems. The author in [23] use 24 API function as features of Ensemble-based classifier for malware detection. They achieved 92.31 as true positive rate and 22.73 as a false positive rate, which is a very high percentage. Besides, Ensemble-based classifier consumed in their experiments on average 1.5s, 13s, and 62.5s for 100, 1000, and 5000 malware, respectively.

In [24], the authors introduced the Anubis platform for dynamic analysis of malware. They launched it online for public usage to receive and analyze user's files. The malware was collected during around two years from submitters. The authors examined the malware dataset and divided its activities according to several aspects, including file, network, registry, botnet, and windows GUI activities. In addition, they discussed the techniques that enabled some malware to detect the sandbox environment. They mentioned tow levels of detection techniques, which are instruction level and API level. Instruction level detection techniques are achieved by monitoring CPU instruction to differ between real CPU instructions and emulated ones. Whereas API level detection techniques are done by recognizing the API functions and system call of Sandboxes.

Ki et al. in[25] proposed a novel approach for malware detecting using dynamic

analysis that was relying on API call sequences. They used VirtualBox and Windows XP as an environment to execute the malware files and record malware’s APIs sequences. Then, they apply DNA sequence alignment through ClustalX application that supports multiple sequence alignment (MSA) to extract common API sequences patterns of malware functions. After that, they categorized the APIs into 26 groups according to the library of MSDN. To obtain the common pattern of API sequence for malware and benign, they used the longest common subsequences (LCSs) and considered it as malware and benign signatures. Also, to reduce the false-positive ratio, they removed common signatures between malware and benign. They also defined critical API sequences that are frequently used by malware to detect unknown malicious programs and activities. The researchers achieved 0.998 of accuracy and 0.999 of F-measure (for the detection).

In [26], the author introduced a scanner approach based on analysis of API sequences for detecting polymorphic malicious executable. They used the PE parser to analyze and determine the API sequence and comparing it with the signatures in the database to find the similarity rate between them.

The authors of [27] analyzed the malware using API call sequences by extracted the frequent itemsets on those sequences. They separated their malware analysis framework into three steps. The first step is API call abstraction that aimed to extract the API sequences using dynamic analysis of malware executables and represented them using unified representation (MIST). Then, the frequent itemsets are abstracted by using algorithms of frequent pattern mining, and the last step is the analysis step, which used to calculate the similarity among various binaries of malware by using a

distance function. They evaluated their method by clustering. The experiments of clustering expressed that their method reduced the computation time with acceptable accuracy.

In [28], the authors tried to understand the malware behavior by extracting the sequence of API calls. They used a static analysis technique on PE of malware. As a result of this research, they grouped the features of API calls into six groups based on their malicious behavior. Those features are Read or write files, copy or delete files, search files to infect, get file information, change file attributes, and move files. Also, they notice that the read/write files were mostly used by the malware to hit the program.

2.3 Hash Based Matching

Approximate Hash-Based Matching (AHBM) has emerged and has been used in several computer industries and academic researches. Because AHBM is not integrated with the existing digital investigation tools, and knowledge of the technique is still limited, AHBM has not gained mainstream popularity. In brief, AHBM is used to identify similar versions of files and fragments recovered from deleted files or memory [1].

Cryptographic hash functions or block-based hashing are the most famous generation of hashing that were designed to find similarities (integrity) between different binary data [2]. Cryptographic hash functions are used for many security purposes and applications such as digital signatures, authentication, Message Authentication

Code (MAC), one-way password file, virus detection, intrusion detection, to construct a pseudorandom function (PRF), pseudorandom number generator (PRNG). It is efficient and fast, but the similarity result will be strongly affected if there is a trivial change on a bit in one of the compared files [29].

Almuhammadi and Bawazeer studied the performance and security of most famous cryptographic hash functions (block-based hashing) and discussed the features of them. They mentioned that the most secure block-based hash function should change around 50% of its digest if there is a small change in the source file. So, the similarity will be lost due to this change. Therefore, this type of hashing (yes or no) is not suitable for using it to detect malware based on API calls because it can not tell to what extent the similarity between malware and signatures. [29].

Introduced in 2006 by Dr. Kornblum, SSDEEP is even more robust than block-based hashing. The idea of SSDEEP relies on breaking up the input file into several chunks through a rolling hash function. Then, a different hash function will be used for hashing all chunks and which produces a tiny digest for each chunk. And the signature will be built depending on the concatenation of hashes for all of the individual chunks [2][3][4].

In 2007, Roussev et al. proposed a new approach called multi-resolution similarity (MRS) hash that tried to reduce the time-consumption of large-scale data processing. It is used to automate the extraction of valuable information from that data in a more efficient manner. This approach was used to compute hashing similarities between binary data. A process of context discovery was used to hash the input data into a series of bloom filters after splitting them into variable-size chunks. Also, by changing

the condition of the context discovery out of a single parameter, a whole family of hashes was generated at various granularity levels [2].

After continued research, Roussev prepared a new approach to reduce false positive and negative rates by extracting statistically improbable features from binary data that have a low entropy scheme. He efficiently attained fingerprint representations of statistically improbable features [30] and put them into the bloom filters. This helped him to get around 99% correct classification, and the misclassification rate did not exceed 0.55%. Besides, one of the advantages of using this tool was the reduction of the required space for similarity hashing for a large-scale dataset. The generated similarity digests would not exceed 2.6% of original data so that it could maintain similarity digests of data up to 1.5 TB in the RAM [6].

An evaluation study on similarity hashes was conducted in [4]. The researcher found that the SSDEEP correlation algorithm relied on the existence of large common data in contiguous chunks. Where SDHASH demonstrated a highly correlated of similarity scores, this is because of its ability for filtering and sorting to find the common data. As a result of evaluation experiments between SSDEEP and SDHASH, Roussev found that SDHASH, on average, had better performance, accuracy, and scalability regarding different sizes of targeted files than SSDEEP.

The three ABHM operation modes are defined in [1] by Bjelland et al: clustering mode, streaming mode, and searching mode. When they used SDHASH in the streaming experiments, it expressed proper monitoring fit for the continuous data stream, like network traffic, to trace interesting data. Also, the clustering experiments on a malicious dataset of PDF documents showed that the techniques of approximate

matching had the potential for classifying and organizing unknown data. For the searching mode, the authors used the AHBM on a large dataset of emails to search for a group of emails. Compared to other searching techniques (e.g., string searching), the AHBM is better due to increasing the effectiveness and reducing the risks of human error through results investigation.

In 2019, the authors of [31] introduced the AHBM approach as a new technique for malware traffic analysis. They used the AVClass tool for malware ground truth labeling. Then, they presented their algorithm for clustering the malware traffic based on the AHBM approach. They achieved 93% and 60% for precision and recall, respectively, when they considered the error rate of AVClass. In general, their work showed promising results to use the AHBM approach as a new technique for malware traffic analysis.

In summary, we discuss the types of malware detection analysis, which are static, dynamic, and hybrid. Then, we go through a set of references related to the malware detection using system events. Finally, we present the literature of hash-based matching.

CHAPTER 3

EXPERIMENTAL SETUP AND DATA COLLECTION

In this chapter, the procedures of data collection are presented, and the newly developed environment, namely, ProPeBox, is introduced. Besides, the malware dataset is shown, and the ground truth labeling steps are explained.

A meticulous experimental evaluation for detecting malware behavior in the computing-world environment is crucial. The virtual machine (VM) applications provide a realistic emulation of real-world operating systems. So, VM lets malware to manifest their real behavior as if malware would execute in the original environment. They should, thus, be capable of exploiting the resources of host machines as well as network services with the least interruptions. Therefore, we can investigate and analyze the full recorded of malware infection traces in a real computing world environment.

We developed a new malware detection environment, and an original Windows 32-

bit OS was installed in it. This is to get results like systems in real-world computing that can be infected by malware. The security and other system settings should as well match the real system environments. Also, Malware 32 bit datasets are selected from VirusShare and VirusTotal. Procmon tool is used for recording system, and malware events and PExec is utilized to automate the traces collecting processes. We wrote python codes to automate and manage all stages of operations in the environment. The purpose of this setup is to allow us to collect real run-time malware behavior and to get an accurate evaluation of the malware detection system. We have conducted all collecting experiments on the cloud virtual machines of the King Fahd University of Petroleum & Minerals (see table 3.1).

Table 3.1: Host Machines Specification for The System and Malware Events Collecting Experiments

Processor	Intel(R) Xeon(R) E5520 @ 2.27 GHz 2.26GHz
RAM	16 GB
Host Operating System	Window 7 Professional 64bit
HDD	300 GB

3.1 Experimental Environment: ProPeBox Environment

For our experiments, we have developed a new, isolated, protected, configurable, secure, and controlled environment to collect run-time behavior of malware samples and as well as clean system events as benign traces, and we called it ProPeBox Environment. The name is inspired by the tools, which are Procmon, PeExec, and VirtualBox,

that have combined by python script to build the environment. It produces a report file contained all host-based system events for a certain period as defined in configuration settings.

VirtualBox (VB) [32] was installed as a protected and controlled virtual machine to execute the malware executables. Oracle VM VirtualBox is an Open Source or pre-built Binaries for Linux, Windows, Solaris, and Mac OS. It is a powerful Virtualization Cross-platform Software for x86-based systems.

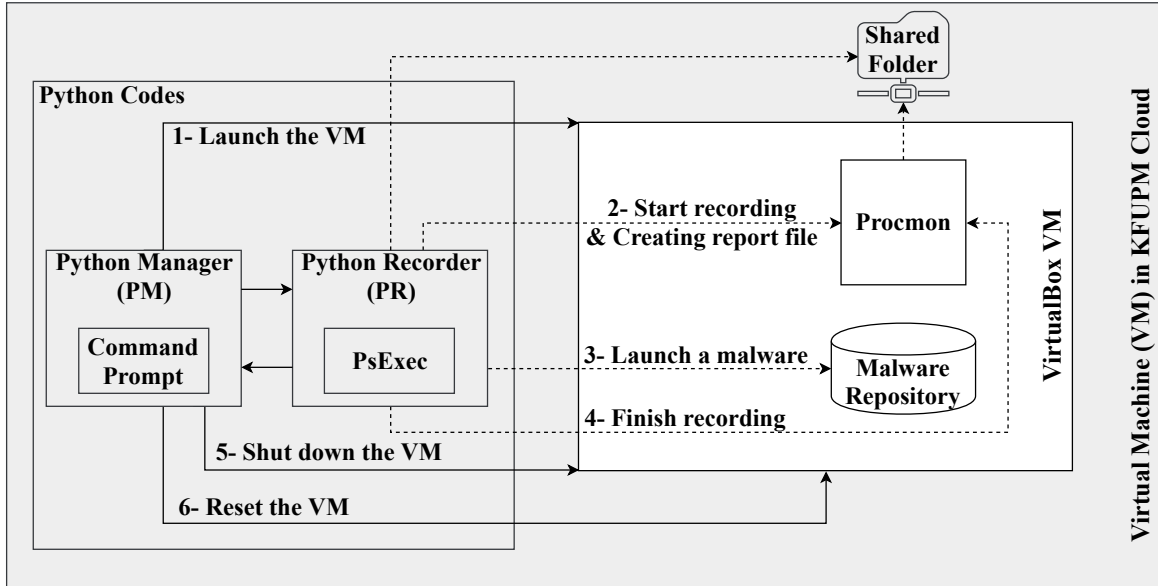


Figure 3.1: Structure of ProPeBox Environment

And for recording the events of malware samples, we utilized a Procmon [33] tool. Process Monitor, aka Procmon, is a free and advanced tool, one of the Windows Sysinternals tools, combines two earlier tools, RegMon and FileMon. Process Monitor monitors, records, and displays in real-time all process/thread, file system, and Registry activities on Windows OS. It is used in application debugging, computer forensics, and system administration. It can filter the captured information on spe-

cific conditions and values such as processes, process IDs, and operations. Besides, it expresses how malware and other applications use Dynamic Link Libraries (DLLs) and other Windows OS resources.

Then, to get control over the recording traces processes, the host-only network is created between the host and the VM to allow PsExec [34] to perform processes remotely. PsExec is also one of the Windows Sysinternals tools. It gives us the ability to execute operations on other systems. So it allows us to start recording system traces, including malware samples events, and terminate that process in a specific amount of seconds. Figure 3.1 shows the structure of ProPeBox Environment.

3.1.1 Experiment Automation

To automate the data collection process of system events and API calls of each malware in the repository, we have written python scripts to manage that process for every malware. We have two python scripts worked together, the first one is used to manage the VB VM, and the second one is utilized to handle the PsExec, Procmon, and malware repository. We called the first one Python Manager (PM) and the other one Python Recorder (PR). All python scripts are executed in the host VM to control the execution of the ProPeBox Environment. The automation process starts with PM. It controls starting ProPeBox VM, transferring the execution instructions to PR, shutting down, and resting ProPeBox VM. In contrast, PR takes control of launching the Procmon, creating the report file, launching a malware from the repository, terminating the recording process. Figure 3.2 illustrates the automation process for recording system events using ProPeBox Environment.

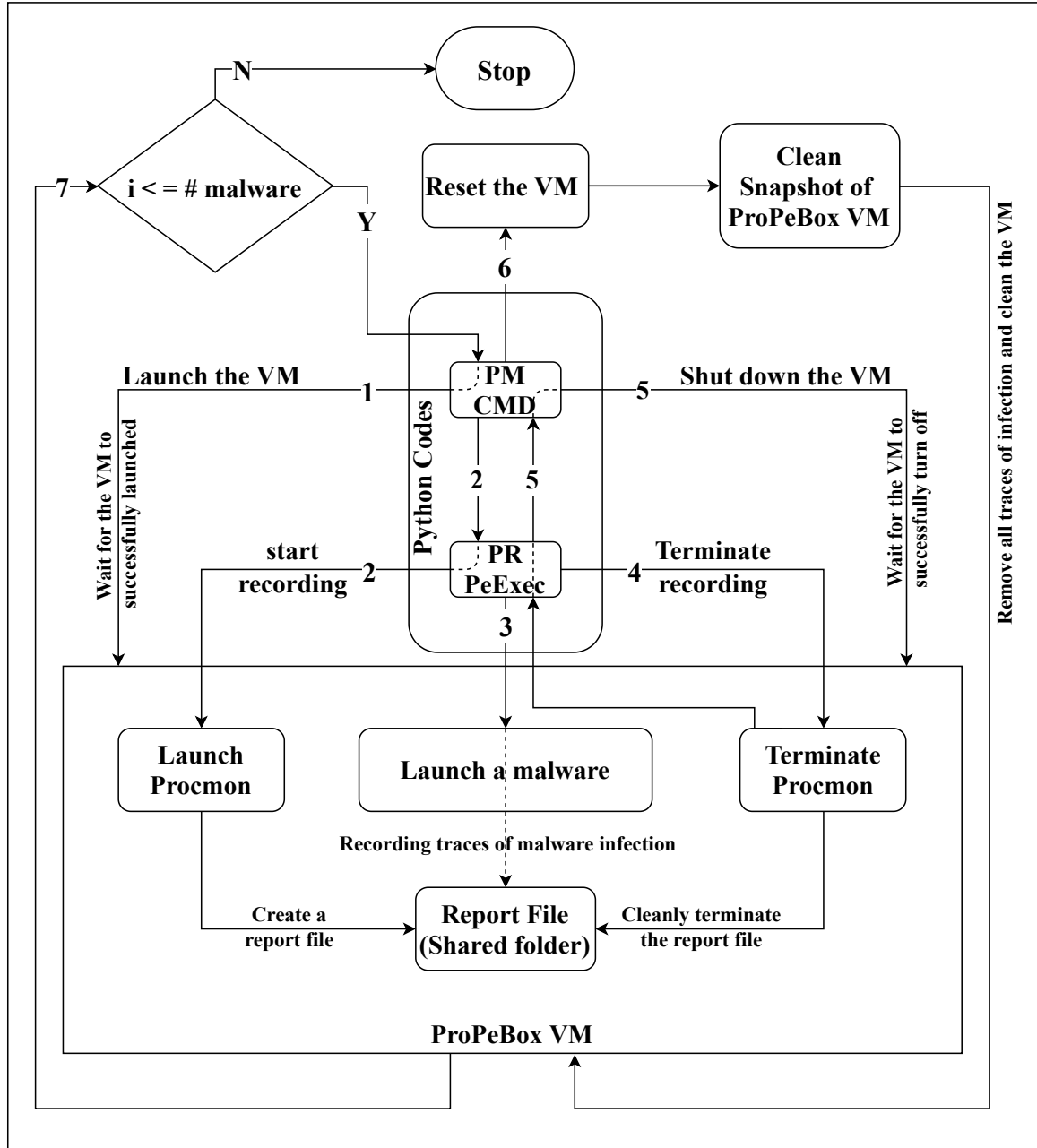


Figure 3.2: Automation Process for Recording System Events Using ProPeBox Environment

The scenario of recording the system events and API calls, including the malware traces, starts when the VM is launched by PM. PM waits for a period to ensure the VM is started normally. Then, PM transfers the execution instruction to PR. And PR sends an instruction to start the Procmon along with creating a report file in the

shared folder. Procmon will start collecting all host-based system events and save them in a report file. The waiting period here is also required to allow Procmon to launch normally. We have chosen a shared folder to save the report file because it is not affected by resetting the VM at the end of the collecting process. After that, PR sends another instruction to launch a malware sample from the repository. At this point, the Procmon tool is already working and recording all system events behavior along with malware traces. Here, we wait again for a while to capture the system event and API calls. Next, the PR sends a new instruction to terminate the collecting process of Procmon to stop recording and cleanly close the report file. Then, after a new waiting for terminating Procmon and report file, the execution is transferred again to PM. PM shuts down the VM and waits for assuring that. Lastly, PM will reset the VM using a snapshot of a clean copy of VM to clean the VM by removing all traces of the last malware sample. This process is repeated continuously with the same queue of operations until the end of the malware counter of PM is reached. Furthermore, PM is used Prompt Command to execute their processes, whereas PR is used PeEXec for that purpose as well. In addition to that, waiting periods can be experimentally configurable based on the host machine capabilities. Moreover, the report file is produced in Procmon’s extension (.PML), which contains information about system and malware traces.

3.1.2 Converting PML Report Files to CSV

CSV data files are considered as one of the most suitable files for malware dataset because it can be handled easily using PANDAS [35]. PANDAS is a python library de-

signed for analysis and data manipulation. It provides operations and data structures for time series and manipulating numerical tables.

As we mentioned before, the ProPeBox Environment produces a report file in PML extension each time. And to prepare the dataset files for PANDAS, we have written python code to convert the PML files to CVS files by exploiting the command prompt instruction supported by the Procmon tool. So, after this point of processing, we have CSV dataset files of system and malware events (see 3.3).

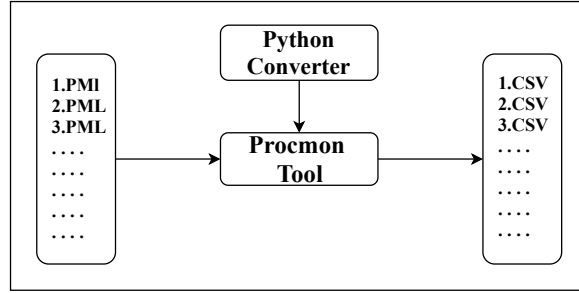


Figure 3.3: Converting PML Report Files to CSV

3.2 Data Collection

Tracing real-time malware events and API calls is a crucial process since it reveals the behavior of malicious activities of malware, expresses how malware can infect the system and can be used to generate malware signatures for malware detectors. In this section, the collected malware datasets, API calls datasets and the ground truth labeling are described.

3.2.1 Malware Dataset and Collection Procedures

We obtained two groups of malware executables. The first group of executables is gathered from the previous work of [31]. Qasem got the malware collection from the public VirusShare [36] repository, excluded all non-Windows 32 bit, and ended up with around 1000 malware of windows 32 bit. We selected randomly 450 executables of them. The second group is taken from the VirusTotal [37] repository after getting special access permission on it. We selected 1500 malware of Windows 32 bit randomly.

After that, we collect the malicious dataset of both groups separately by executing malware binaries using ProPeBox Environment. We collect system along with malware events of 10 samples for each malware in round-robin fashion. To be clear, the number of malware and system events is significantly huge. It can exceed hundreds of thousands of events in a matter of seconds. So, we have decided to collect the events and API calls of malware for only 20 sec. However, the whole process using ProPeBox Environment lasts around 3 minutes, including collecting the malware events and API calls. As we mentioned before, ProPeBox Environment contains several stages, and each stage takes a waiting period to be executed. The waiting period is configurable and is based on host machine capabilities. Table 3.2 shows the time parameters of different stages of ProPeBox Environment.

To collect malicious activities of malware, we have implemented the experimental setup in 3 and 3.1. We have started collecting in February 2019 with the VirusShare malware group. We used a VM on the KFUPM cloud as a host machine, installed the ProPeBox on it, and created a host-based Network between them. And to accelerate

Table 3.2: Time Parameters of Different Stages of ProPeBox Environment

Stage	Time	Note
Launching the VM	90 sec	It can be taken less in a powerful machine.
After launching the Procmon tool	5 sec	To ensure that the Procmon is started.
Collecting malware events	20 sec	Period of collecting malware activities.
Terminating Procmon & closing the report file	20 sec	To guarantee that the report file is cleanly closed.
Shutting down the VM	20 sec	To ensure that the VM is closed.
Resetting the VM	20 sec	To ensure that the VM is cleaned and the malware infection traces are removed.

the collecting process for the VirusTotal malware group, we have dedicated three VM in the KFUPM cloud as a host machine, installed the ProPeBox in each VM along with creating a host-based network between them. The malicious activities of the VirusTotal malware group are collected between April and May of 2019. For the VirusShare dataset, we ended up with 4480 samples for 448 malware, whereas, for the VirusTotal dataset, we ended up with 14190 samples for 1419 malware. As we expected, some of the malware is not working in the ProPeBox Environment. This could be because of malware using anti-VM, anti-debugging, and anti emulation techniques to prevent themselves from detecting or reverse engineering in the virtual environment. In some cases, malware could be successfully executed, but the Procmon tool does not cleanly close the report file due to the potential of latency in the host-based network between the host machine and the ProPeBox Environment. This causes delaying termination Procmon and closing the report file as well. Therefore, when the waiting period of terminating the Procmon process has ended, the shutdown instruction of ProPeBox

using PM script takes place. At that point in execution, the environment will be closed without cleanly closing the report file so that the report file will be damaged and the collected data will be lost. So, all corrupted data and unexecuted malware samples have been excluded. At the end of the collecting process, we have got a PML report files dataset of system and malware events. Therefore, all PML report files are converted to CSV files to prepare them for the filtering process using PANDAS that extracts the events and API calls of malware samples.

3.2.2 Malware Events and API Calls Dataset

When we get the dataset of infected systems by malware events from 3.2.1, and to get only the dataset of malware events and API calls, we have to filter the dataset of infected systems by removing all other events and API calls that do not belong to either the malware process(es) or their children's process(es). To do that, we have developed a python script to look for and count the malware Processes ID (PID) and their children for each malware sample. Then, we have written python script to filter the report file of the infected systems and extract all events and API calls of the malware sample. The filtering and extracting operations are done based on malware's processes, along with their children's processes.

The malicious dataset contains valuable data about the malware behavior that we got recorded by the Procmon tool. The collected data can be divided into three main groups, application details, event details, and process management data. Application details of malware involve static data determined when the process of malware started up. It includes:

- Malware Process Name.
- Description, Version, and Company Name of malware executable: These are the data of the file version module extracted from the file's image.
- Image Path and Command Line: These are the command line and the full path of the started process.
- Malware Executable Architecture: all malware on our dataset is 32 bit.

The most attractive data is included in the event details group. It contains the most valuable information about the dynamic behavior of malware events and API calls, such as:

- Malware Event Classes: The malware events can be classified into five main event classes, which are process (Process Start, Load Image, Process Create, Thread Create, etc.), file system (WriteFile, ReadFile, CloseFile, CreateFile, CreateFileMapping, DeviceIoControl, etc.), registry (RegCloseKey, RegCreateKey, RegDeleteKey, RegDeleteValue, RegSetKeySecurity, RegQueryKey, etc.), network (TCP Connect, TCP Disconnect, TCP Receive, UDP Receive, UDP Send, etc.), and profiling.
- Malware Operations: Malware operations are the API calls that are performed and requested by malware.
- path of malware operations: The path is the location of a file, DLL library, registry, IP address, etc., which used by the malware operation.

- Category malware operations: The malware operations on a file system or registry can be categorized into one of four types, Read, Write, Read Metadata, and Read Metadata.
- Result: It is the result of malware operations. It can be SUCCESS, NAME NOT FOUND, ACCESS DENIED, etc.
- Duration: It is the period for performing the malware operations.

Event details group of malware also includes other data like Date & Time, Sequence Number, Relative Time, Time of Day, and Completion Time. The last group of events is the process management group. It includes run-time information about malware such as Process ID, Thread ID, Parent ID, User Name, Session ID, Authentication ID, Integrity, and Visualized.

3.3 Ground Truth Labeling

The malware ground truth labelings are taken by a third-party tool. In this thesis, we use AVCLASS [38] as ground truth labeling for our malware dataset. AVCLASS is an open-source and automatic labeling tool for malware. It depends on the VirusTotal AV label report file of malware for labeling it. The VirusTotal AV label report file, which is a JSON file, contains the result of malware labeling using many AV engines supported by VirusTotal. Therefore, AVCLASS accepts the JSON file as input and then labeling the malware according to the majority of labeling votes of AV engines. If there is no preponderance agreement on a malware labeling between AV engines,

AVCLASS labels that malware a SINGLETON.

We collected the JSON file for our datasets from the VirusTotal and implemented the AVCLASS tool on them. We have got 48 families for the first dataset of 448 malware from the VirusShare. Thirty families include only one malware, and 12 of them are labeled as SINGLETON. In the VirusTotal dataset, we have got 79 families from 1419 malware. Fifty-two families include only one malware, and 26 are labeled as SINGLETON. The tables 3.3 and 3.4 below show the malware families of VirusShare and VirusTotal datasets.

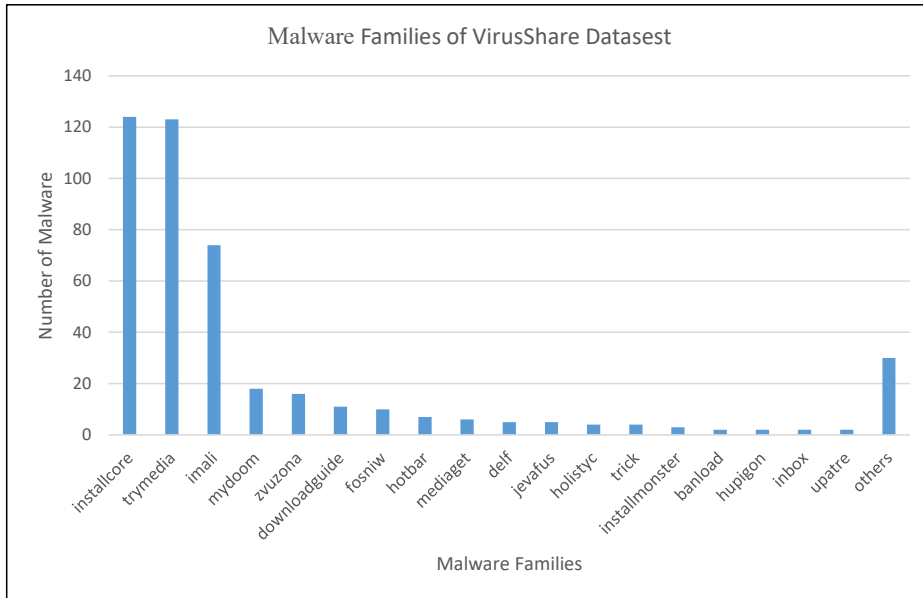


Figure 3.4: Families of VirusShare Malware Samples Dataset

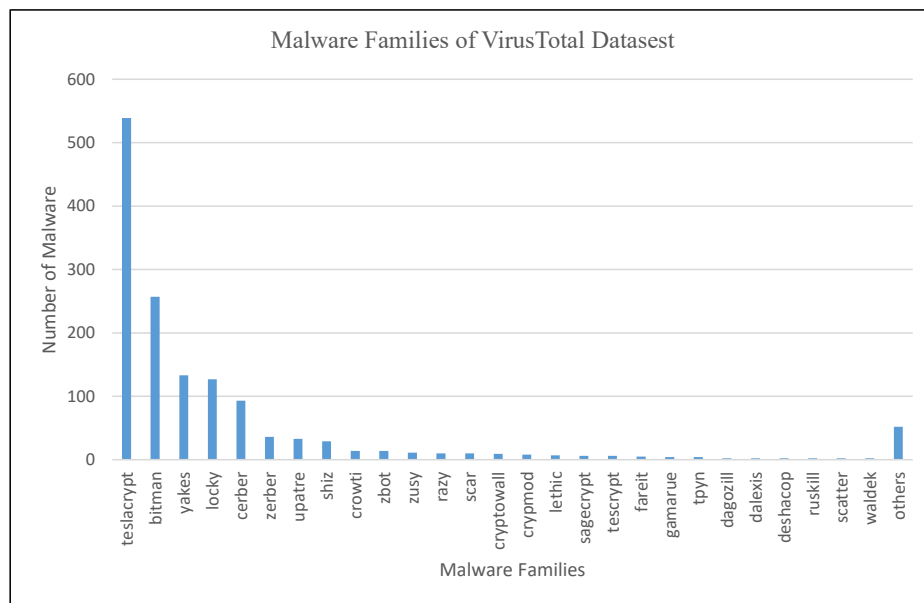


Figure 3.5: Families of VirusTotal Malware Samples Dataset

Table 3.3: Malware Families of VirusShare Dataset

Family Name	# Malware	Family Name	# Malware
installcore	124	midia	1
trymedia	123	multiplug	1
imali	74	ngrbot	1
mydoom	18	onlinegames	1
zvuzona	16	pcoptimizing	1
downloadguide	11	proinstall	1
fosniw	10	qghost	1
hotbar	7	sefnit	1
mediaget	6	SINGLETON_1731...	1
delf	5	SINGLETON_1e94...	1
jevafus	5	SINGLETON_3c72...	1
holistyc	4	SINGLETON_53bd...	1
trick	4	SINGLETON_729f...	1
installmonster	3	SINGLETON_832e...	1
banload	2	SINGLETON_8b10...	1
hupigon	2	SINGLETON_8e3d...	1
inbox	2	SINGLETON_9623...	1
upatre	2	SINGLETON_a5ce...	1
agobot	1	SINGLETON_c5af...	1
albot	1	SINGLETON_c8ad...	1
autoit	1	vespula	1
blackhole	1	vilsel	1
directdownloader	1	zbot	1
fenomengame	1	-	-

Table 3.4: Malware Families of VirusTotal Datasest

Family Name	# Malware	Family Name	# Malware
teslacrypt	539	genericcryptor	1
bitman	257	godzilla	1
yakes	133	kovter	1
locky	127	midie	1
cerber	93	myxah	1
zerber	36	necurs	1
upatre	33	nemucod	1
shiz	29	pwszbot	1
crowti	14	rack	1
zbot	14	SINGLETON_1218...	1
zusy	11	SINGLETON_132a...	1
razy	10	SINGLETON_2373...	1
scar	10	SINGLETON_29e4...	1
cryptowall	9	SINGLETON_2a55...	1
cryptmod	8	SINGLETON_2dbe...	1
lethic	7	SINGLETON_308c...	1
sagecrypt	6	SINGLETON_3d2a...	1
tescrypt	6	SINGLETON_41c4...	1
fareit	5	SINGLETON_52ea...	1
gamarue	4	SINGLETON_71ee...	1
tpyn	4	SINGLETON_7ebd...	1
dagozill	2	SINGLETON_9dea...	1
dalexis	2	SINGLETON_9f8d...	1
deshacop	2	SINGLETON_a44e...	1
ruskil	2	SINGLETON_b35b...	1
scatter	2	SINGLETON_b719...	1
waldek	2	SINGLETON_b8ee...	1
beebone	1	SINGLETON_bbb8...	1
carberp	1	SINGLETON_d513...	1
critroni	1	SINGLETON_e077...	1
delf	1	SINGLETON_e25c...	1
dmalocker	1	SINGLETON_ea75...	1
dorkbot	1	SINGLETON_ebb2...	1
eetz	1	SINGLETON_f549...	1
enestaller	1	SINGLETON_f73c...	1
enestedel	1	teerac	1
expiro	1	tinba	1
fraudrop	1	zboter	1
fsysna	1	zegost	1
gdsda	1	-	-

CHAPTER 4

APPROXIMATE HASH BASED MATCHING (AHBM) APPROACH

Approximate Hash Based Matching is presented in this chapter. Firstly, we explain the approach motivation and then give a technical discussion about how we can use the AHBM for malware detection. And then, several sections, including AHBM signature generating, criteria of feature selection, and digest similarity comparison, are introduced.

4.1 Approach Motivation for Malware Detection

Hash functions are well-known techniques for detecting similarities between objects (files, binaries, etc.). In digital forensic, the similarity issue is not only related to the objects themselves. It also is associated with the content of objects, even if they are

in different types or extensions. However, block-based hashing is the most famous hashing that can be used to identify the binary data similarity. It is fast and computationally efficient. But if a small part of an object's content is changed or reordered, the similarity score will be missed. So to address the hash fragility problem and the object's blocks order changing, the Approximate Hash Based Matching (AHBM) is considered as an accurate and scalable solution [4] for block-based hashing problems. Comparing with the literature of malware detection, AHBM is simpler, scalable, and efficient for disk space requirements [6]. In fact, the similarity score of AHBM is measured depending on the string matching of objects. Where it considers the objects as strings of bytes and generates the similarity rate (0-100) by matching the sequences of their signatures.

To overcome the block-hashes issues, AHBM extracts the signatures of objects without needing to separate them into blocks. AHBM is used data fingerprinting [39] to choose the distinct features of data objects and then to find a correlation between objects based on them. Equally important, Winnowing and K-gram are implemented to address the hash problem of block alignment of objects. The idea of using K-grams is that the string of bytes of the data object is split into contiguous substrings and overlapping them with k-size known as k-grams. Then, the k-grams are hashed, and a part of them is picked as the fingerprint of the object. The effective algorithm of using and selecting hashes of k-grams is Winnowing. Winnowing algorithm uses a sliding window cross over the string of bytes of the data object and then selects a representative feature from each window.

Considering the AHBM is mainly relying on string matching, we use the sequence

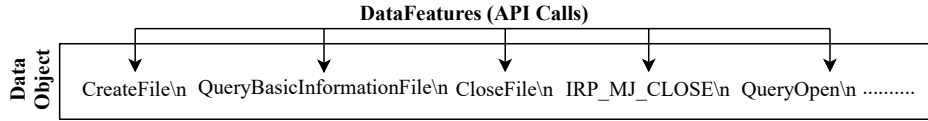


Figure 4.1: Data Object and Features (Real API Calls)

of API calls of malware as the data object of malware. Each API call of the data object is called a data feature. And because of that, we use a subset of the series of data features of malware as a malware signature by implementing the Longest Common Substring (LCS) technique on malware the data object. The proposed malware detection system is depending on the winnowing and data fingerprinting techniques that inspired by the AHBM approach of Roussev in [6], [30], [40]. Roussev introduced the AHBM tool, namely, SDHASH, in 2010. It is used to generate the fingerprinting of API calls of malware samples, and their signatures based on selecting the statistically-improbable features that are least likely to transpire in different objects. Then, the bloom filters [41], which very efficient approach for disk space utilization, is used to store the hashed selected features. At this point, each object is stored in one bloom filter or more. Thus, the similarity measurement is done by comparing the bloom filters of malware samples against their signatures.

4.2 AHBM Signature Generation

4.2.1 Criterion of Features Selection for API Calls of Malware

In this section, the usage of the AHBM approach will be discussed to show how we used it to express the feature selection criteria and generate the signatures of the malware API calls dataset. As we explained before, for each window slide cross over

API calls of malware, AHBM is looking for the statistically improbable features that are least likely to appear in other data objects of malware. So, picking statistically unlikely occur features is the main criterion of the data features selection process. On the other side, the selected features should represent the generality of data objects of malware.

Table 4.1: Malware API Calls vs Representative Characters

API_Calls	RC	API_Calls	RC
CloseFile	A0	RegCloseKey	E0
CreateFile	A1	RegCreateKey	E1
CreateFileMapping	A2	RegDeleteKey	E2
⋮	⋮	⋮	⋮
ProcessCreate	F5	TCPConnect	G0
ProcessExit	F6	TCPDisconnect	G1
ProcessStart	F7	TCPReceive	G2
⋮	⋮	⋮	⋮

In our case, we have recorded the API calls of ten samples for each malware. And because of the variation of API names, we have built a representative set of characters table to replace every API call to its alias from that table (see table 4.1). This step guarantees that all data features (API calls) are represented in the same length (size) of characters. Hence, the chosen sample size of data features from the data object each time will be the same for all malware samples as well as for their signatures. In our case, the sample size parameter is 3, that is because each one feature contains 2 letters and a new line (see figure 4.2). Besides, the sliding window of the AHBM technique will choose the same number of data features each time. Therefore, the statically improbable features have been selected in the right way without any interference

between the contiguous data features of API calls for all malware samples, and the Bloom filters will be generated using a proper and correct way.

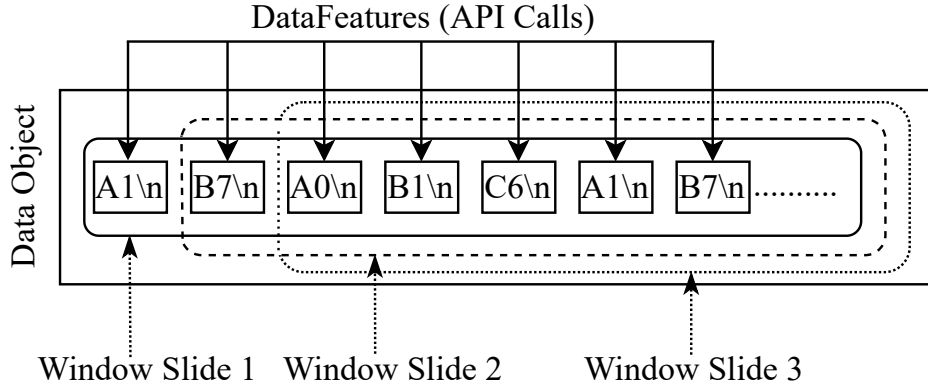


Figure 4.2: Data Object and Features Using Representative Set of Characters

Now, to identify statically improbable features of malware API calls, Roussev in [6], [30], [40] used Shannon entropy to measure the amount of information and uncertainty level in data objects. Suppose that we have a sequence of malware API calls (M) as a data object. It contains a sequence of data features (API calls) $\{X_1, X_2, \dots, X_n\}$, the entropy in the first-order level computed as

$$H(M) = - \sum_{i=1}^n P(X_i) \log_2 P(X_i) \quad (4.1)$$

Where $P(X_i)$ is the empirical probability value of a data feature X_i in the malware data object string.

Then, to release the entropy value from the feature size (B) and make it independent, the normalization of the entropy score is computed as follows:

$$H(M)_{norm} = \lfloor 1000 \times H(M) / \log_2 B \rfloor \quad (4.2)$$

Where all malware data features are put in 1000 equivalence classes with neglecting the feature-length (B). Then, from the representative set of H_{norm} , the empirical probability distribution is assigned as a precedence value V_{prec} for each feature of malware API calls. It is a value between 0 to 1023. This value indicates that the probability of the data features in the data object. This means that the smaller V_{prec} value the common H_{norm} is, the larger V_{prec} value the uncommon H_{norm} is.

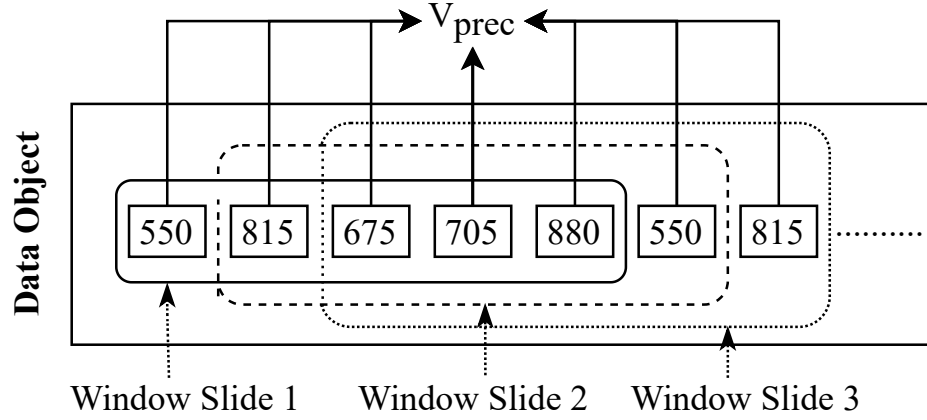


Figure 4.3: Data Object and Features

After that, the popularity value V_{pop} is determined to reveal the relation between the V_{prec} of data features with their neighbors. To compute the popularity value V_{pop} , for all data features in each sliding window (W), the leftmost data feature that has the lowest V_{prec} is identified and incremented by one (where $V_{pop} \leq W$). Consequently, the statistically improbable data features get higher values of V_{pop} than other data features. In our case, based on our empirical experiments, we have chosen 64 for the window sliding (W) parameter.

At this point, we put 0 value as a V_{pop} for all data features to prepare them for the next step. The threshold (t) for selecting the statistically improbable of malware API calls features is defined as ($0 \leq t \leq W$). A sweep process is done using a sliding

window (W) across all data features of malware data object to increment the value of V_{pop} for selected V_{prec} by one. At the end of this process, the threshold (t) is used to determine all chosen data features. That is, all data features that have a popularity score higher than or equal to the threshold will be selected ($V_{pop} \geq t$).

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
3	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0
4	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0
5	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0
6	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0
7	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	5	0	1	0	0	0	0	0	0	0	0	0	0
8	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	5	0	1	0	0	0	1	0	0	0	0	0	0
9	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	5	0	1	0	0	0	2	0	0	0	0	0	0
10	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	5	0	1	0	0	0	3	0	0	0	0	0	0
11	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	5	0	1	0	0	0	4	0	0	0	0	0	0
12	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	5	0	1	0	0	0	5	0	0	0	0	0	0
13	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	5	0	1	0	0	0	5	1	0	0	0	0	0
14	Vprec	550	815	675	705	880	520	800	675	815	880	705	505	520	675	715	800	550	705
	Vpop	1	0	0	0	0	5	0	1	0	0	0	5	1	0	0	0	1	0

Figure 4.4: Example Explained Popularity Value V_{pop} Calculation Using Sliding Window of Size 5 and Data Sample of Size 18

Figure 4.4 shows the calculation process for popularity value V_{pop} of a sample of data features of size 18 with a sliding window of size 5. The sliding window ($W = 5$) starts with the first five precedence values V_{prec} and selects the value in the first position because it has the lowest value of V_{prec} and is the leftmost feature in the sliding window as well. Consequently, the popularity score for that V_{prec} will be increased by one. Next, the sliding window will move one step to cover the values

of precedence from the second value to the sixth value. Here, the lowest value is in the sixth position, and it will be selected and increased from step 2 until step six for the same reason. This process will be continued to the last step in the position of 18th. Finally, the threshold (t) value will be the crucial condition for selecting the statistically improbable features. Assume that $t = 4$, this means that only two features will be fulfilled the requirement ($V_{pop} \geq t$), which are in the sixth and twelfth positions. In our case, based on our empirical experiments, we have chosen 8 as a threshold (t) for selecting the statically improbable features.

4.2.2 Generating Fingerprints and Similarity Digests

The next step after selecting the statically improbable features of the data object is to create the representation fingerprint using hash digests. Even though block-based hashing does not produce a false positive rate, but it is not efficient for space representation. As well as, it can not efficiently handle the membership queries; that is, it can not tell if an object is partially similar to others.

One of the most efficient techniques for set representations is Bloom filters. Bloom filter is a fixed size (m) of bit-vectors used efficiently to store set representations with a minimal amount of false-positive rate. All bits in the Bloom filter are by default set to 0. Suppose that, from the previous step, we have a set of elements of chosen statistically improbable features from the data object of malware $\{a_1, a_2, a_3, \dots, a_n\}$. To insert a new element a_i to the Bloom filter, it will be hashed using k different hash functions $H_1, H_2, H_3, \dots, H_k$. The digest of a_i is split into k sub hashes. After that, for each sub hash, the 11 least significant bits are considered as an address (index) to

insert it inside the Bloom filter, and the corresponding bit is changed to one. Similarly, membership queries are treated in the same manner, instead of changing the bits to 1, it will be tested to examine if they already exist inside the Bloom filter or not. If all k bits already exist and equal 1, then they are already represented and no need for new insertion. Otherwise, they will be inserted in the same manner. However, there is potential by chance for the query to set inside a Bloom filter during checking of their bits; this will lead to a false-positive (FP) result. Fortunately, the probability of FP can be controlled and quantified by determining the number of inserted elements in the Bloom filter. It is important to know if all bits of hashed entries are set to 1, then there is no 100% guarantee for that to be in the representation set of the Bloom filter. Whereas if there only one computed entry k is set to 0, then we can tell that it is not found in the representation set.

Note that, before inserting a new feature, the filter checks if it is already represented, the feature is not added again into the filter, and the counter of embedded elements is not increased. So, the same features are prevented from inserting into the Bloom filter more than once. Therefore, the false positive rate is reduced by avoiding the duplication of features representation in the filter, and the similarity accuracy is increased as well [6].

For our experiments, SHA-1 has hashed the selected features and produces 160 bits digest. The digest is split into five sub hashes ($k = 5$) with a size of 32 bits; the 11 least significant bits of each one of them is used as filter's index. Furthermore, each one of the sub-hashes is treated as an independent hash to insert the feature inside the filter (see figure 4.5). According to our implementation, the selected features are

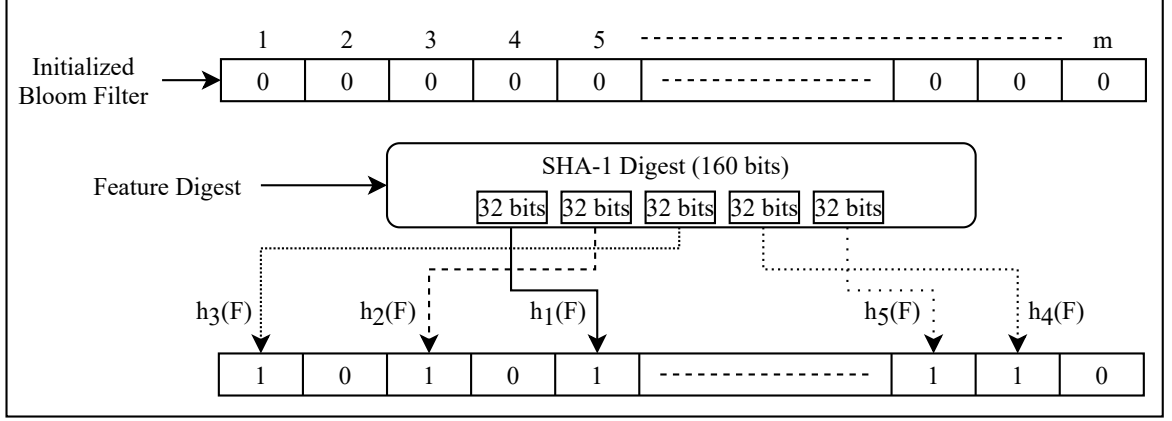


Figure 4.5: Process of Inserting Features Inside Bloom Filter

represented in Bloom filter(s) of the size of 256 byte ($m = 256 \text{ bytes} = 2048 \text{ bits}$) and the maximum numbers of elements per filter is 128 elements ($n = 128$). Broder et al. used the previous data (k, m, n) in [42] to calculate the false positive rate and maintain it under a reasonably low threshold. As they mentioned in their research, the false positive rate f could be computed using the following formula:

$$f = \left(1 - e^{-\frac{kn}{m}}\right)^k \quad (4.3)$$

In our case, the false positive rate for Bloom filter implementation is around 0.00139.

In summary, at this point in our experiment, the signatures are generated by five independent 32 bits sub-hashing of SHA-1 and stored the represented set (RS) (similarity digest) in a space-efficient manner using concatenation of Bloom filters as follows:

$$RS = s \mid bf_1 \mid bf_2 \mid bf_3 \mid \cdots bf_s \quad (4.4)$$

Where:

- S : number of Bloom filters.
- bf_i : the i th Bloom filter stored a representation set of malware API calls.

4.3 Similarity Comparison of AHBM Digests

After generating the digests of data objects and representing them into Bloom filters, the next step is to compute the detection rate (similarity score) by comparing the Bloom filters of the malware and its signature. For this purpose, the comparison process is implemented by detecting the similarity of the commonality bits between the Bloom filters of data objects.

Assume that we have two Bloom filters of size (m) ; one of them for the API calls (data object) of malware signature (bf_{sig}), and the other one is for the malware data object (bf_{mal}). Let (n_{mal}) and (n_{sig}) is the number of features of (bf_{mal}) and (bf_{sig}), respectively, where $(n_{sig} < n_{mal})$, and (n_{sm}) is the number of common features between both of them. And as we mentioned before in 4.2.2, (k) is referred to how many times SHA1 are independently used. (e_{sig}, e_{mal}) and (e_{sm}) are indicated to the number of bits set to one in (bf_{sig}) , (bf_{mal}) , and $(bf_{sig} \cap bf_{mal})$, respectively.

To find the Bloom filters similarity, we follow the same steps and values of derivation in [6]. The Similarity score of Bloom Filters (SBF) can be computed, as shown below:

$$SBF(bf_{sig}, bf_{mal}) = \begin{cases} -1 & \text{if } n_{sig} < N_{min} \\ 0 & \text{if } e_{sm} \leq C_t \\ \left[100 \frac{e_{sm} - C_t}{E_{max} - C_t} \right] & \text{otherwise} \end{cases} \quad (4.5)$$

where,

- N_{min} : It is the minimum number of features in Bloom filters needed to compute meaningful SBF value. According to [6] as we follow, N_{min} is 6 features for a Bloom filter of size 256 byte contained 128 features.
- (e_{sm}) : It is indicated to the number of bits set to one in $(bf_{sig} \cap bf_{mal})$.
- C_t : It is a cutoff threshold in which all matched bits are assumed due to chance.

It is defined as shown below:

$$C_t = \alpha(E_{max} - E_{min}) + E_{min} \quad (4.6)$$

- E_{min} and E_{max} : The estimation of the minimum (E_{min}) and the maximum (E_{max}) of matching bits by chance can be calculated as follows:

$$E_{min} = m(1 - p^{ke_{sig}} - p^{ke_{mal}} + p^{k(e_{sig} + e_{mal})}) \quad (4.7)$$

$$E_{max} = \min(kn_{sig}, kn_{mal}) \quad (4.8)$$

- $(p = 1 - \frac{1}{m})$: It is a probability that a single bit still 0 after setting a single bit.

- α : It is a calibrating parameter. It is set experimentally to value of 0.3 that gives us 0 as the similarity score for Bloom filters of unrelated malware data objects.

The Similarity score of Bloom filters in 4.5 shows three cases for the similarity score between two representation set of Bloom filters. In the first case, if the number of features in the Bloom filters of signature is not enough (in other words, less than N_{min}), the similarity score will be equal to -1 . The second case is considered for the similarity score that can be occurred due to chance. So, if the shared bits between signature and malware filters that set to one are less than or equal to the cutoff threshold C_t , the similarity score goes to 0. However, in the third case, when the features in the Bloom filters of signature are higher than N_{min} and the common bits set to one in both representation set of filters are higher than C_t , the similarity score (1 - 100) is estimated by $\left[100 \frac{e_{sm} - C_t}{E_{max} - C_t}\right]$.

Now, after preparing the Bloom filter computation, the similarity score calculation of data objects of malware and its signature is taken place. Let *sig* be a data object of malware signature, and *mal* be a data object of malware API calls. Let $bf_1^{sig}, bf_2^{sig}, \dots, bf_s^{sig}$ and $bf_1^{mal}, bf_2^{mal}, \dots, bf_m^{mal}$ be set of Bloom filters computed from *sig* and *mal*, respectively. Assume that ($s < m$), the Similarity Score (SS) between data objects of API calls of malware and its signature is computed as follows:

$$SS(sig, mal) = \frac{1}{s} \sum_{i=1}^s \max_{1 \leq j \leq m} SBF(bf_i^{sig}, bf_j^{mal}) \quad (4.9)$$

As defined in 4.9, the similarity score of two Bloom filters comes from comparing

the Bloom filter of signature, which is the shorter one ($s < m$) with the Bloom filter of malware API calls. The similarity computation starts by comparing the first Bloom filter in the malware signature (bf_1^{sig}) with all Bloom filters in the malware ($bf_1^{mal}, bf_2^{mal}, \dots, bf_m^{mal}$) and records the highest similarity score as a result of the first comparison. This process repeatedly continues by comparing each Bloom filter in the signature against all Bloom filters of malware API calls and takes the maximum score of similarity each time. Finally, the average of all recorded similarity scores will be considered as the similarity score SS between sig and mal . And it is a rate between 0 to 100.

Note that the larger size of the Bloom filter of mal does not affect or reduce the similarity score. As well as having all bits of sig occur in separated filters in mal does not generate a high similarity score.

In this chapter, we have discussed the methodology that we follow to generate and represent the malware API calls and their signatures sets using a space-efficient manner with Bloom filter. As well as, we show the mathematical model for computing similarity score between the malware and signatures.

4.3.1 Selecting The Threshold Value for AHBM

As we mentioned before, the AHBM tool generates a value between 0 to 100 as the detection rate between two data objects. Therefore, we need to define a threshold value for our detection system to let us decide if the signature data object in the malware detection system detects the malware or not ($0 \leq t_{score} \leq 100$). For this purpose, and according to our extensive experimental testing on the training dataset

as well as by considering the results of empirical experiments conducted in [6] and [31], we select the threshold score t_{score} value of 21.

In this chapter, we introduce the motivation of using AHBM for malware detection and explain how we can adapt it to work on the API calls along with illustrated examples. After that, we have discussed the technical parts of the AHBM and selected the criteria based on extensive experiments.

CHAPTER 5

MALWARE SIGNATURE GENERATION USING THE LONGEST COMMON SUBSTRING (LCS)

In this chapter, we discuss and explain the procedures of generating the malware signatures, including extracting the malware signatures by using the Longest Common Substrings (LCS), filtering and cleaning the findings of LCS, and choosing the threshold and parameter values for our experiments.

5.1 Longest Common Substring (LCS)

Longest Common Substring (LCS) is a similarity measure used to find the longest common consecutive string between two strings or more. It differs from the Longest

Common Subsequence, where the latter is utilized to find only the consecutive symbols (characters) in two or more strings. For example, assume that we have two strings $S1 = \text{"ABBBABAA"}$ and $S2 = \text{"ABBABABA"}$; the Longest Common Substring (LCS) of $S1$ and $S2$ is "BBABA" with a length of 5 whereas the Longest Common Subsequence of both of them is "ABBA" with a length of 4. Therefore, we have used the Longest Common Substring (LCS) to find the longest common sequence of API calls amongst all samples of each malware and consider it as malware signatures. In most cases, we find that the extracted LCS strings contain a good representation of various API calls executed by malware. As well as the signatures of LCS sequences have enough length to allow the AHBM to select the statistically improbable features and store them in Bloom filters to use the signatures for detecting the malware based on their API calls activities.

5.2 Simplified Method of LCS for Extracting Signatures of Malware

Before implementing the LCS algorithm on our malicious dataset, we analyze the cases of getting the LCS for all ten samples of each malware. We have observed that the optimal case can occur when all samples of malware are executed the same sequence of API calls in each sample. However, the lengths of recorded API calls of malware samples vary according to the state of the infected system during the execution time of collecting system events. So, the maximum length of the extracted signature using the LCS algorithm in the optimal case will be as long as the length of the smallest

sample of malware. And in the other cases, the signature's length will be less than the length of the smallest sample of malware. Based on this fact, we have suggested a simplified method for extracting the common API calls sequences using the LCS algorithm.

```
F7F8F4F4A1.....G8G2E7E7F1A1A1A1D1D3D2
F7F8G8F4F4.....A1A1A1D1D3D1D2D9B7A0F9
F7F8G8F4F4.....E7F1A1A1A1D3D2D9B7
F7F8G8F4F4.....F1A1A1A1D1D3D2D9B7A0F9G2E7E7F1A1A1A1D3D2D9B7A0F9G8G8G8G8G8
F7F8G8F4F4.....F9G2E7E7F1A1A1A1D1D3D2D9D1
F7F8G8F4F4.....D3D2D9B7A0F9G8G8
F7F8G8F4F4.....A1A1A1D3D2D9B7A0F9G2E7E7F1A1A1A1D1D3D2D9B7A0
F7F8F4F4A1.....A1D1D3D2D9D1B7A0F9G2E7E7F1A1A1A1D3D2D9B7A0F9G8
F7F8G8F4F4.....E7F1A1A1A1D1D3D2D9B7A0F9G2E7E7F1A1A1A1D3D2
F7F8G8F4F4.....A1D3D2D9B7A0F9D9B7A0D1F9G8
```

Figure 5.1: Example Illustrated The Length of API Calls of All Samples of One Malware

The idea of the simplified method for using LCS on the API calls of malware samples is to apply the LCS algorithm between the smallest malware sample against others of each malware. So, the cost of the execution time when using the LCS on all samples at the same time is reduced. Besides, the most crucial step to reduce the time complexity of the LCS algorithm is to use the representative set of characters instead of using the real names of malware API calls. For instance, assume “CreateFile”, “CreateFileMapping”, and “DeviceIoControl” are a part of API calls for a malware sample and “A1”, “A2”, and “A3”, respectively, are the corresponding representative set characters of them. Now, we are able to use the representative string of “A1A2A3” for LCS algorithm instead of using the real API sequence “CreateFileCreateFileMappingDeviceIoControl”. In this example, the length of the real sequence of API calls is

reduced by 700% by using the representative set of characters (see table 4.1).

5.2.1 Implementation of Simplified Method of LCS Algorithm

We have written python scripts to apply the LCS algorithm between each sample against others for every malware. The purpose of this step is to prepare all possible signatures for the cross-validation evaluation technique that we are used to evaluating the malware detection system. As it is known, the LCS algorithm is time-consuming. The time complexity of implementing the LCS algorithm in the worst case is quadratic time ($O(n^2)$) and linear in the best case ($O(n)$) [43]. So, we have created an account on the Hyper Performance Computing (HPC) machine of Information and Communications Technology Center (ICTC) at King Fahd University of Petroleum & Minerals (KFUPM). We have used the HPC account to execute the simplified method of the LCS algorithm on our dataset. Table 5.1 illustrates some specifications of HPC machine at ICTC of KFUPM.

Table 5.1: Specification of Hyper Performance Computing Machine at KFUPM

# of Processor	20
CPU Cores	10 cores (for each processor (10*20))
CPU Family	6
Model	62
Model Name	Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
RAM	128 GB

As a result of implementing the simplified method on the HPC, the execution time for the LCS algorithm based on the simplified method is significantly decreased. Table

5.2 shows the execution time of the LCS algorithm based on the simplified method. In this case, the LCS algorithm takes around a week to generate the longest common sequences for the VirusShare dataset and approximately twenty days for obtaining the LCS strings for the VirusTotal dataset.

Table 5.2: Time-Consuming for Generating Signatures Using LCS Algorithm Based on Simplified Method

Dataset	Days	Hours	Minutes	Seconds
VirusShare	6.383495	153.2039	9192.23	551534
VirusTotal	19.719109	473.258	28395.52	1703731

The result of using the LCS algorithm based on the simplified method on all the ten samples of each malware is a text file containing nine strings. Each one of them is considered as the longest common API call sequence that produced by applying our LCS script between the API calls of a malware sample against other samples of the same malware. All strings are manipulated through several processes, including filtering and cleaning the LCS Strings. We have applied this process for all samples of training data sets of malware (80% for training and 20% for testing), and at the end of this process, we will get eight text files for each malware (text file per training sample). These files that contained the signatures will be used to select the malware signature for the detection process by using the cross-validation technique (see section 6.2).

5.2.2 Filtering Process of The LCS Strings

While applying the simplified method of the LCS algorithm, many of the obtained strings contain an unwanted excess number at the beginning of a string or letter at

the end of a string or in both positions. As we explained in 5.2, each API call is replaced by a representative set of characters that contain a letter (A, B, ...) with a number (1, 2, ...) (see table 4.1). Therefore, it is expected to find one or more of LCS strings starting with a number or ended with a letter produced by taking the LCS of the representative set of characters of a malware sample against the representative set of characters of another malware sample.

☐ 1	A1E2E1F7G0G2A1
☐ 2	2A1E2E1F7G0G2A1
☐ 3	A1E2E1F7G0G2A1B
☐ 4	2A1E2E1F7G0G2A1B
☐ 5	E2E1F7G0G2
☐ 6	2G0G2A1E2E1F7G0G2A1G0G2G1B
☐ 7	2A1E2E1F7G0G2A1B
☐ 8	A1E2E1F7G0G2A1
☐ 9	E2E1F7G0G2

Figure 5.2: Example Illustrated The Findings State of The LCS Strings Based on The Simplification Method

Assume that we have the LCS strings of a malware sample against others of the same malware using our simplified method illustrated in figure 5.2. There are five strings containing unwanted excess characters; the second string starts with a number, which is an excess number as well as the last character in the third string. Similarly, the fourth, sixth, and seventh strings start and end with excess characters. These unwanted characters are filtered and removed from all LCS strings (see figure 5.3).

With the result of the filtering process, the list of LCS strings is ready for the

cleaning process.

5.2.3 Cleaning Process of The LCS Strings

The cleaning process is implemented to reduce the number of generated strings from the LCS algorithm based on the simplified method by removing the redundancy between LCS strings of signatures. Note that the cleaning process must only be applied to the LCS strings of the training data set for each fold. So, the LCS strings related to the testing data set will be removed before starting the cleaning process.

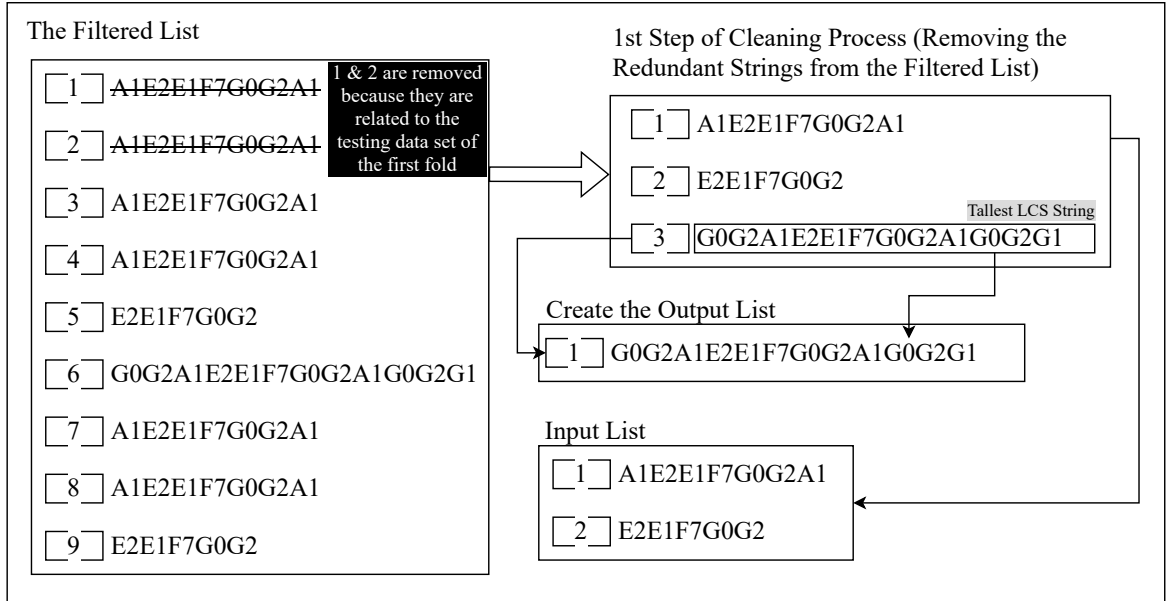


Figure 5.3: The Results of Filtering Process on The Findings of LCS Strings Based on The Simplification Method and Preparing The Input and Output Lists for Cleaning Process

The first step in the cleaning process is to find the identically similar strings in the LCS list and remove the redundant strings. Then, we create a new list called the output list and add the tallest LCS string among all filtered strings for the same malware sample to the output list and removed the tallest string from the LCS list

(input list). We choose the longest string of LCS list because it represents the majority of API calls of the malware that mostly appeared in all strings of the LCS list. Now, we have two lists the output list and the input list. The next step is to compare each string in the input list with all strings in the output list. If a string from the input list is found in a part of any string in the output list, the string will be removed from the input list. Otherwise, the string will be added to the output list and removed from the input list. This process is continued until removing all strings from the input list. Consequently, the results of the cleaning process for each malware are stored in the output list (see figure 5.3).

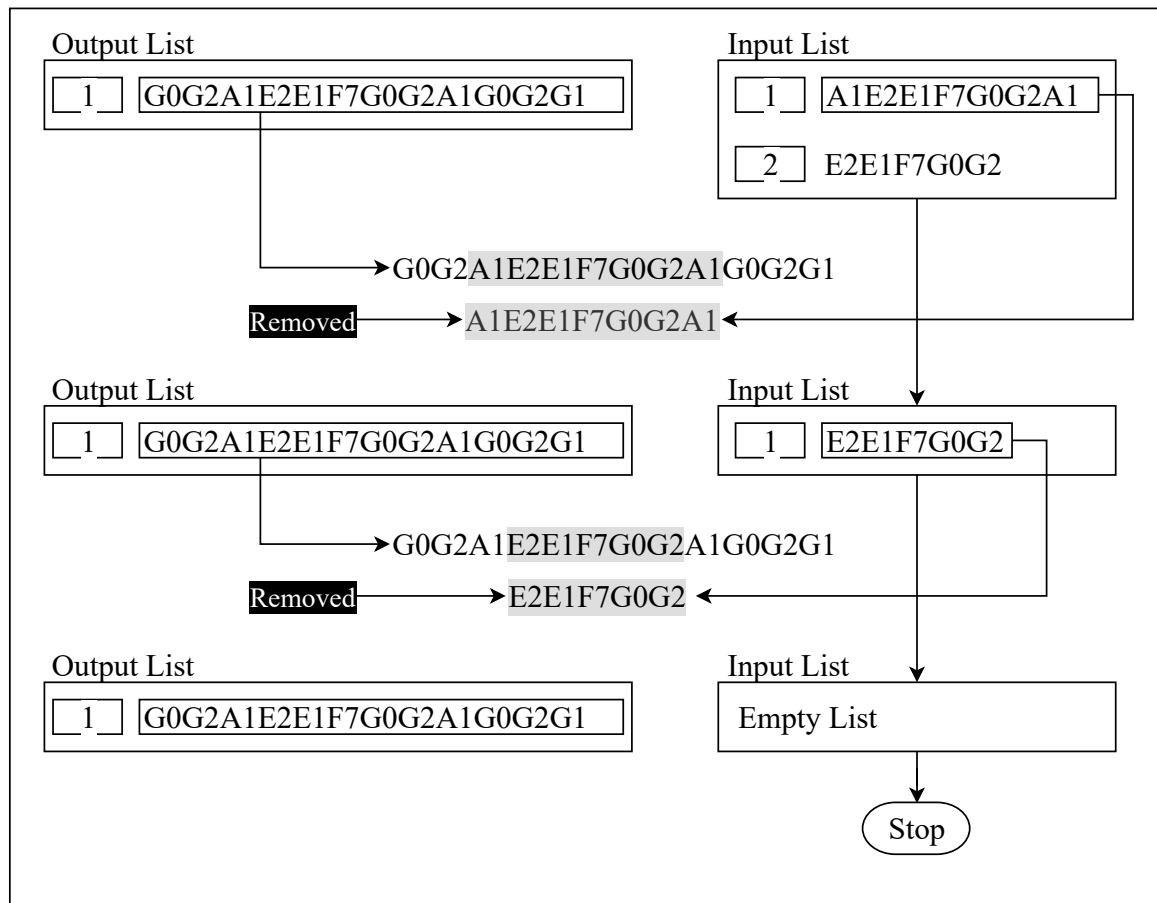


Figure 5.4: Cleaning Process Procedures

The rest of the example goes as follows (see figure 5.3), firstly, the filtered list is cleaned by removing the redundant strings. Strings in the third, fourth, seventh, and eighth positions in the filtered list are identically similar, so one of them will be selected and remain in the input list, and the others will be removed. As well as the strings in the fifth and ninth positions are treated in the same way. The string in the sixth position will be added to the input list as is since it is not identically similar to other strings. Hence, we have done the first step of the cleaning process, and we get the filtered list without redundancy in its strings. After that, we create a new list called the output list and add the tallest string in the input list to the output list. At that point, we have two lists; the output list contained one string, and the input list held the remaining LCS strings after applying the filtering and the first operation in the cleaning processes.

The last step in the cleaning process is to search for each string in the input list inside all strings in the output list. If any sequence from the input list is found as a part of any sequence in the output list, it will be removed because it is already represented. Otherwise, the string that is not seen as a portion of output list strings will be added to the output list. As it is shown in figure 5.5, all strings in the input list are found inside other string from the output list. So it is removed from the input list, and the running script is stopped when the input list is empty. The results of these processes are the representative set of characters that are stored in the output list. In the next section, the criteria for selecting the malware signatures will be discussed.

5.3 The Criteria for Selecting The Malware Signatures Based on The Simplified Method

In this section, we present the criteria for selecting the malware signatures based on the simplified method. As a result of the filtering and cleaning processes, we have got the output lists contained the longest common sequences by implementing the LCS algorithm based on the simplified method. Now, the crucial step is to select the signature of each malware from its LCS output lists (list per training sample). So, the most critical point for choosing the malware signatures is to meet the criteria of feature selection for AHBM approach. This is to ensure that the detection system based on AHBM will be operated correctly with the expectation of a high accuracy of detection. To achieve that, we have to select the signature based on two main criteria:

1. The sequence of features (API calls) in the signature should be long enough to allow the AHBM parameters such as threshold and sliding window to work correctly. Consequently, we should define the threshold value of the length of features represented in the signature to be accepted for our system.
2. The signature sequence should be contained as many as possible of different features to allow the AHBM to select the statistically improbable features amongst them. Therefore, the representative set of features of the signature is selected easily and represented in the Bloom filter with a great expectation of high accuracy of detection rate. So, it is vital to determine the threshold number of the representative set of distinct API calls in each signature.

Accordingly, the exclusion process will be implemented to exclude any malware that does not meet the criteria.

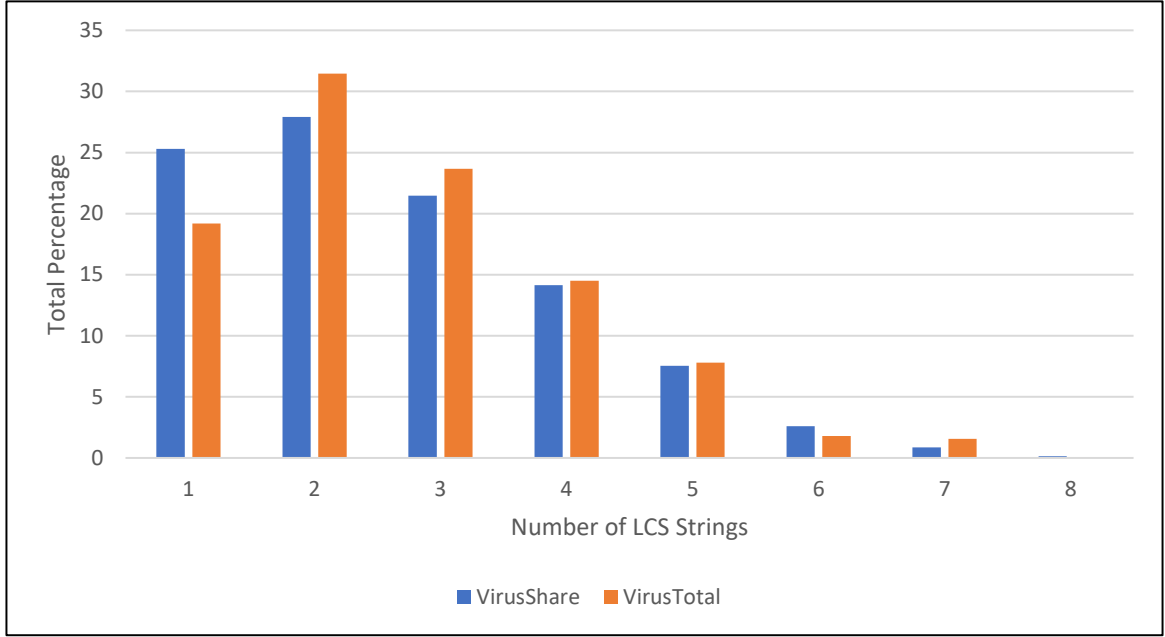


Figure 5.5: Statistics of LCS Strings Based on Simplified Method

As a result of the filtering and cleaning processes for LCS strings, we select the signature of each malware in our datasets from their LCS output list by considering the selection criteria. Figure 5.5 shows the percentage of the number of strings in the output lists for VirusShare and VirusTotal datasets. The first group of the output list contains only one sequence of API calls. It accounts for the quarter of the VirusShare dataset and around the one-fifth of the VirusTotal dataset of the total number of output lists. Consequently, each sequence in this group considers as the signature of its malware. The other groups of output lists contain more than one sequence, so the criteria for selecting the signatures are implemented to select one sequence of them as a malware signature. The chosen thresholds values for selecting signatures criterion are discussed in the next section 5.4. More discussion about signature selection is

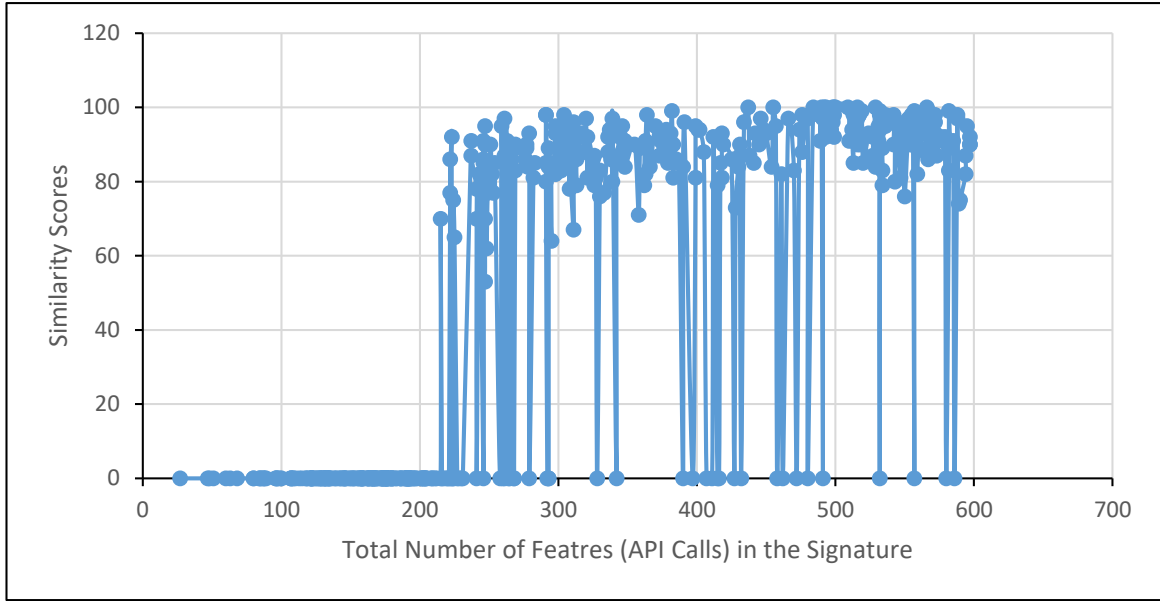
presented in the next chapter.

5.4 Thresholds Values for Selecting Signature Criteria

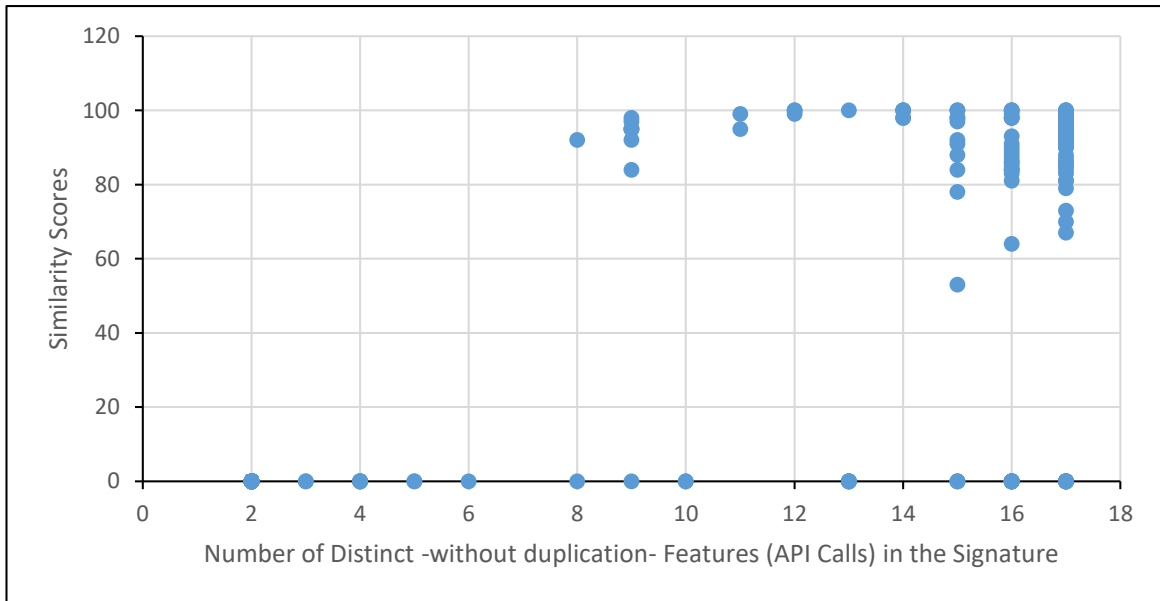
As mentioned in section 5.3, we have to choose two threshold values for malware signatures. The first threshold is the minimum length of features that allow AHBM to work correctly (the minimum number of features in the whole signature sequence). And the other threshold is the minimum distinct number of features in the signature string (how many distinct features -without duplication- in the string).

To determine the thresholds of malware signatures, we have conducted comprehensive experiments on the training datasets. And based on our experimental evaluation, we find that the signature should contain in length more than 230 features of API calls, and it should be built at least from 8 different features. Therefore, we excluded any malware if its signature does not contain 8 or more distinct features as well as if the total number of features in the sequence is less than 230 features (see figure 5.6 a and b).

Moreover, in the included set of signatures, we noticed the length of the signature sequence is more affected than the number of the distinct API call in the signatures. We observed that the signatures that have more length and an acceptable number of API calls achieve high detection rate than the signatures that have less length and more number of API calls.



(a) Selecting Signature Threshold Value for The Minimum Length of API Features



(b) Selecting Signature Threshold Value for The Distinct Number of API Calls in The Signature String

Figure 5.6: Thresholds Values for Selecting Signature Criteria

5.5 Choosing The Malware Signatures Based on The Simplified Method of LCS

As a result of the Cleaning process on the training folds, we have got a list of LCS strings for each malware sample. Now, to select the malware signature through the training samples in each fold, we follow and implement the steps of the simplified method of LCS. For each malware in the training folds, the first step is to determine the smallest sample in length amongst all training samples for the same malware. Then, the output list of the smallest sample generated by LCS based on the simplified method is checked; if it contains one LCS sequence, it will be considered as the malware signatures for all samples of that malware. Otherwise, the output list contains two or more sequences (less than 10 sequences). In this case, we have implemented the criteria for malware selection and select the tallest sequence in the list. But when we found two or more in the same length, we choose the one that the larger number of the distinct API calls. The last probability is to find tow or more signature with the same length and the distinct number of API calls; in this case, we select one of them randomly.

In summary, we have introduced the methodology of generating the malware signatures and how we can optimize the usage of Longest Common Substring (LCS). As well as we have explained the internal processes of using LCS and after getting the LCS sequences, including filtering and cleaning the findings of LCS strings. Finally, we discuss the methodology for choosing the malware signatures.

CHAPTER 6

MALWARE DETECTION BASED ON API CALLS USING AHBM APPROACH

6.1 Malware Detection Model

In this section, we continue explaining the remaining stages of our malware detection model. Also, we discuss the implementation of cross-validation to evaluate our malware detection model. After that, we present and discuss the findings of the malware detection system based on API call sequences using AHBM as well as show the scalability and time efficiency of our model.

We have built our model using four stages; malware dataset collection, generating malware signatures, generating similarity digest, and malware detection using AHBM approach (see figure 6.1).

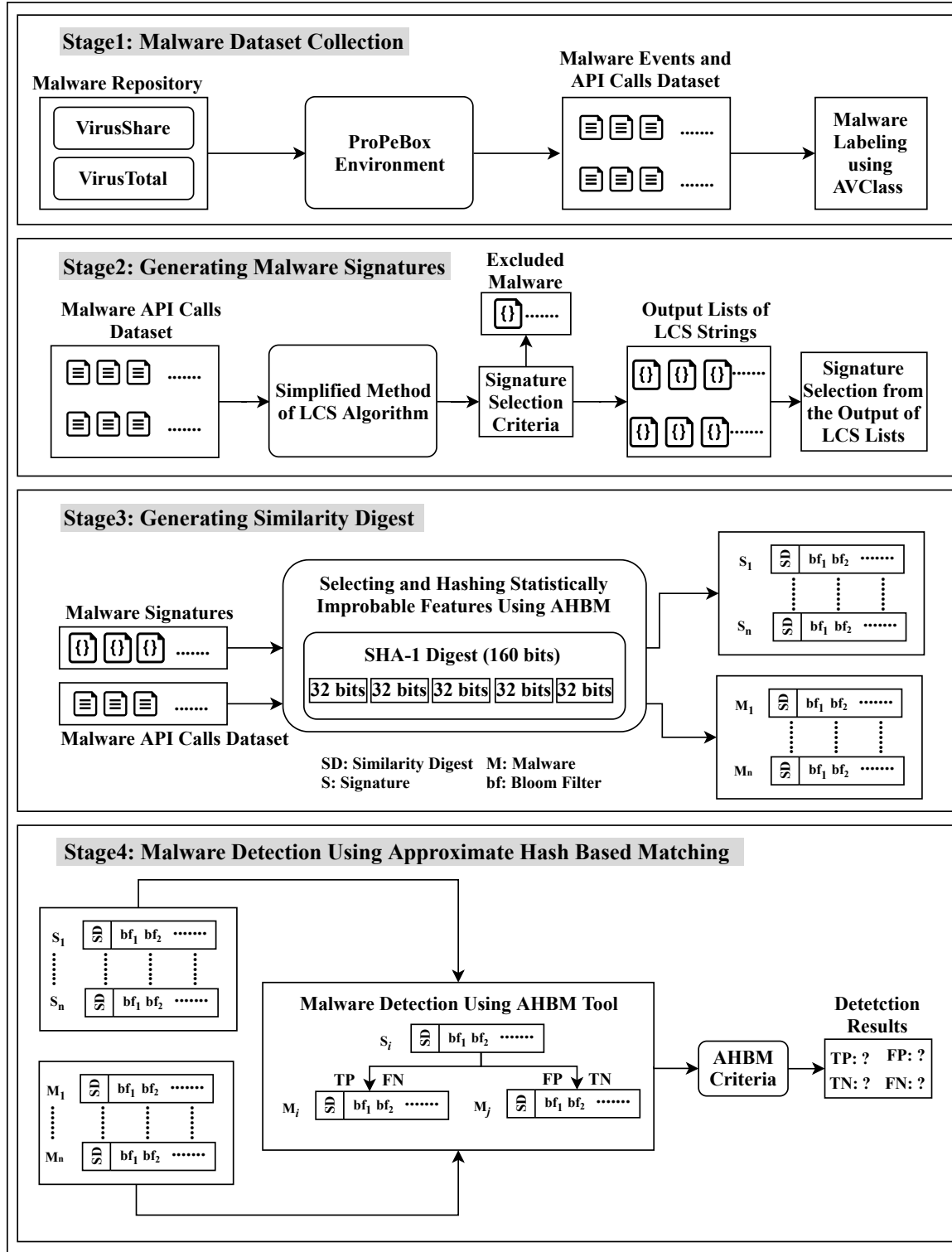


Figure 6.1: Malware Detection Model

Firstly, we collect two datasets of malicious executables from two public repositories, which are VirusShare and VirusTotal. The datasets have collected by our new protected and controlled environment, namely, ProPeBox environment. After that, we use the AVClass tool to label the malicious datasets. At the end of this stage, we have got the datasets of malware events and API calls and prepare them for the next phase. (see chapter 3).

Then, we have presented our simplified method of the Longest Common Substring algorithm to extract the longest common sequence between the training malware samples. We have suggested our simplified method for using the LCS algorithm to reduce the computational time of the LCS algorithm. Then, we put two criteria for selecting the malware signatures based on our empirical experiments. As a result of this stage, we have got the signatures of each malware in both datasets. (see chapter 5).

In the third stage, as we presented in chapter 4, we have started dealing with the AHBM approach when we discuss the representation process of signatures and malware API calls digests inside Bloom filters. Each malware and its signature are stored independently in one or set of concatenated bloom filters that are used in the next stage to compute the similarity score between them.

In the detection stage, as we mention in the literature review, the API calls are widely used for malware detection systems with diverse techniques such as [44][45][46][47] and others. The proposed approach using AHBM shows promised results malware detection by matching the similarity between the sequences of malware samples and their signatures. Noteworthy, the similarity process is achieved by comparing and finding the commonality bits of Bloom filters of the signature and malware

API calls. This process has been done by comparing each Bloom filter in the signature against all Bloom filters of the malware API sequence. And the maximum score of all comparisons for each Bloom filter has been counted. Then, the detection rate is taken by averaging the similarity scores of the signature's Bloom filters. The length of malware API calls does not dilute the similarity score; at the same time, to get the highest similarity rate, the API calls features should be presented in the malware API calls and concentrated in few numbers of Bloom filters. More demonstrations and results discussion will be shown in the following sections in this chapter.

Figure 6.1 illustrates all stages of our model for malware detection, starting with collecting the malware events and API calls using the ProPeBox environment, passing to generating the malware signatures, creating the similarity digests, and ending with applying the similarity process in the last stage.

6.2 Malware Detection

Most of the existing work on malware detection using cross-validation techniques to evaluate the malware detection process. The main idea of cross-validation is to partition the dataset into k equally size folds. Then, k iterations of validation and training are conducted using a different partition each time.

6.2.1 Cross-Validation Technique

In our experiments, we have partitioned each dataset into 5-folds to implement the cross-validation technique. Each fold contains two parts; training and testing. The

training part represents (80%) out of malware samples, and the testing part represents (20%). The cross-validation process is iterated 5 folds (times) by selecting different parts of samples for testing each time. Table 6.1 shows the chosen samples in each fold for each iteration.

Table 6.1: Training and Testing Samples for Each Malware at Each Fold

Folds	Training Samples	Testing Samples
Fold 1	{3,4,5,6,7,8,9,10}	{1,2}
Fold 2	{1,2,5,6,7,8,9,10}	{3,4}
Fold 3	{1,2,3,4,7,8,9,10}	{5,6}
Fold 4	{1,2,3,4,5,6,9,10}	{7,8}
Fold 5	{1,2,3,4,5,6,7,8}	{9,10}

In our malware detection model, we have implemented the cross-validation technique along with our simplified method of LCS for signatures selection. So, in the training phase of each fold, signatures selection is the first step. For that purpose, we have applied the simplified method of LCS on each training datasets to select the malware signatures. Therefore, for each malware, we choose the smallest sample amongst training samples and apply the criteria of signatures selections on its output list of LCS. Consequently, if the output list has only one signature, it will be considered as the malware signature. Otherwise, we will apply the selection criteria, and one of the output list strings will be selected as the malware signature according to criteria of signatures selection. (see section 5.5).

Noteworthy, we have ended up with 397 malware (3970 samples) in the VirusShare dataset, and 968 malware (9680 samples) in the VirusTotal dataset. The reduction in malware number is due to implementing the criteria of the signatures selection that excluded unsatisfied malware as well as we exclude all unclassified malware by

AVClass that labeled as SINGLETON in both datasets. Besides, the exclusion process contains malware that discarded during collecting malware events. Table 6.2 shows the statistics of malware samples in VirusShare and VirusTotal datasets.

Table 6.2: The Malware and Samples Number of VirusShare and VirusTotal Datasets

Dataset	Folds	Malware	Samples	Training Samples	Testing Samples
VirusShare	5	397	3970	3176	794
VirusTotal	5	968	9680	7744	1936

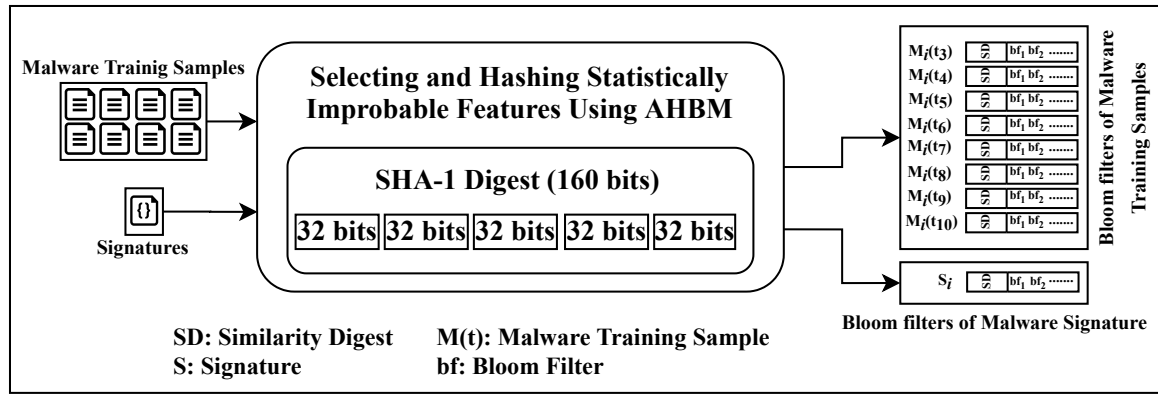


Figure 6.2: Example of Generating Bloom Filters of Malware Training Samples and Signature

Then, the malware detection process is firstly performed by checking the dataset of training samples against their signatures one by one at a time. As we mentioned, each malware has eight samples in the training dataset, so, we training all samples for every malware with its signature. In this step, the AHBM approach generates the similarity digests by producing the Bloom filters of malware samples and the signature. Then, the AHBM approach has been used to calculate the similarity score between each training sample ($M_i(t)$) and its signature (S_i) for each malware by comparing their Bloom filters. This process has been applied to the training phases of each fold. Figure 6.2 The figure illustrates the process of similarity digest generating for malware

training samples and their signatures.

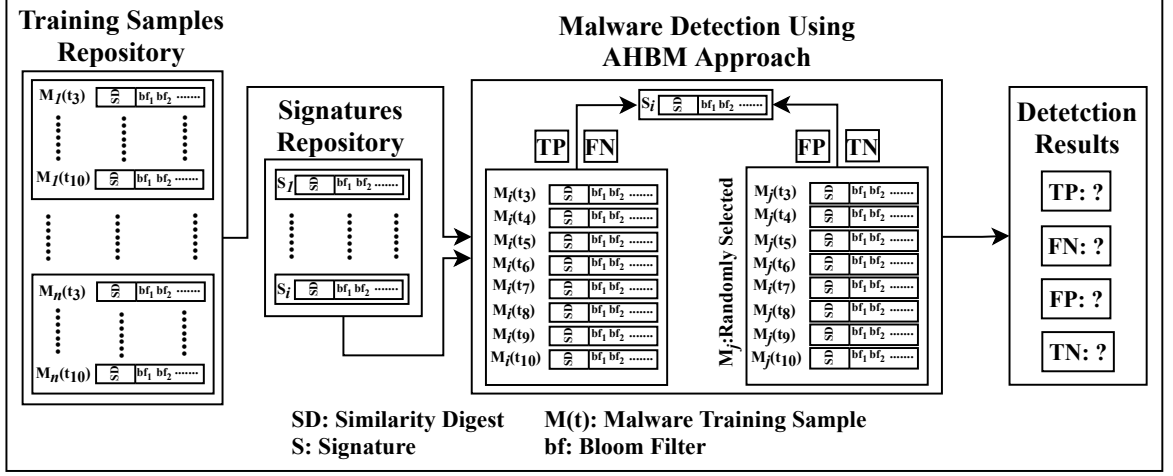


Figure 6.3: Malware Detection Using AHBM Approach

Now, according to the chosen parameter of the detection threshold value, which is 21, the True Positive (TP) is calculated for each sample ($M_i(t)$) correctly detected by its signature (S_i) if the detection rate is equal or more than 21 ($D_{rate}(M_i(t), S_i) \geq 21$). Whereas all wrongly rejected samples with a detection rate of less than 21 ($D_{rate}(M_i(t), S_i) < 21$) are counted as False Negative (FN). Moreover, we need to calculate the False Positive (FP) and the True Negative (TN). For that purpose, for each training malware sample ($M_i(t)$), we consider all other malware samples from other families as benign samples, and we randomly select ($M_j(t)$) which is a malware from a different family of ($M_i(t)$). Then, we have trained the signature (S_i) of ($M_i(t)$) with the training samples of ($M_j(t)$) and check the detection rate of them. If the signature (S_i) identifies any training sample of ($M_j(t)$), this case is counted as an FP because it is incorrectly detected ($D_{rate}(M_j(t), S_i) \geq 21$) by signature from a different family. Otherwise, If the signature (S_i) does not identify any training sample of ($M_j(t)$), this case is counted as a True Negative (TN) because the training sample is

correctly undetected by a signature from a different family ($D_{rate}(M_j(t), S_i) < 21$). (see figure 6.3).

Likewise, the same procedures of the training phase have been followed up to validate and evaluate our malware detection model by checking each malware sample in the testing phase in each fold with the same set of signatures used in the training phase. The detection cases are summarized in the following equation:

$$D_{rate} = \begin{cases} TP & \text{if } D_{rate}(M_i(t), S_i) \geq 21 \\ FN & \text{if } D_{rate}(M_i(t), S_i) < 21 \\ FP & \text{if } D_{rate}(M_j(t), S_i) \geq 21 \\ TN & \text{if } D_{rate}(M_j(t), S_i) < 21 \end{cases} \quad (6.1)$$

6.3 Results and Discussion

Malware detection experiments are conducted for the training and testing phases of each fold in both datasets three times. Then, the average of them is considered as a result of TP, FP, TN, and FN. Tables 6.4, 6.5, 6.6, and 6.7 are shown the results of training and testing for VirusShare and VirusTotal datasets by taking the average of three conducted experiments in each fold for malware detection using the AHBM approach. All experiments are conducted on a laptop having the following specifications.

Table 6.3: Machine Specification for Malware Detection Experiments

Processor	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
RAM	16 GB
Operating System	Windows 10 Home 64-bit

Table 6.4: Training Results of VirusShare Dataset to Evaluate The Malware Detection Model Using Cross-Validation Technique

Folds	Training			
	TP	FP	TN	FN
Fold_1	0.97418136	0.290722082	0.709277918	0.02581864
Fold_2	0.9738665	0.393261965	0.606738035	0.0261335
Fold_3	0.9738665	0.410894207	0.589105793	0.0261335
Fold_4	0.97418136	0.419185558	0.580814442	0.02581864
Fold_5	0.97418136	0.418240974	0.581759026	0.02581864
Average	0.97405542	0.386460957	0.613539043	0.02594458
	0.974	0.386	0.614	0.026

Table 6.5: Testing Results of VirusShare Dataset to Evaluate The Malware Detection Model Using Cross-Validation Technique

Folds	Testing			
	TP	FP	TN	FN
Fold_1	0.97103275	0.308144416	0.691855584	0.02896725
Fold_2	0.97481108	0.399244332	0.600755668	0.02518892
Fold_3	0.97481108	0.416876574	0.583123426	0.02518892
Fold_4	0.97355164	0.400083963	0.599916037	0.02644836
Fold_5	0.97355164	0.403862301	0.596137699	0.02644836
Average	0.97355164	0.385642317	0.614357683	0.02644836
	0.974	0.386	0.614	0.026

The findings of our experiments show promising results for using the AHBM approach in the malware detection field. It also proves that the effectiveness of using our simplified method for extracting the sequences of malware signatures using the LCS algorithm. Besides, the results reveal the capability of the AHBM approach to be used

Table 6.6: Training Results of VirusTotal Dataset to Evaluate The Malware Detection Model Using Cross-Validation Technique

Folds	Training			
	TP	FP	TN	FN
Fold_1	0.946022727	0.306387741	0.693612259	0.05397727
Fold_2	0.945893595	0.307592975	0.692407025	0.0541064
Fold_3	0.946022727	0.302621384	0.697378616	0.05397727
Fold_4	0.946022727	0.3059573	0.6940427	0.05397727
Fold_5	0.946022727	0.307291667	0.692708333	0.05397727
Average	0.945996901	0.305970213	0.694029787	0.0540031
	0.946	0.306	0.694	0.054

Table 6.7: Testing Results of VirusTotal Dataset to Evaluate The Malware Detection Model Using Cross-Validation Technique

Folds	Testing			
	TP	FP	TN	FN
Fold_1	0.944214876	0.301997245	0.698002755	0.05578512
Fold_2	0.946280992	0.309917355	0.690082645	0.05371901
Fold_3	0.945764463	0.305010331	0.694989669	0.05423554
Fold_4	0.946280992	0.303202479	0.696797521	0.05371901
Fold_5	0.945764463	0.309917355	0.690082645	0.05423554
Average	0.945661157	0.306008953	0.693991047	0.05433884
	0.946	0.306	0.694	0.054

for malware detection. It also demonstrates the scalability of the AHBM approach when it deals with a massive amount of malware data objects. However, these results contain a few FN either because of undetectable occurrences or the weaknesses of API distribution in the sequences of signatures. In some cases, the signatures meet the criteria of signatures selection but the API calls of the signatures are not distributed in the whole sequence of signatures. For example, when we check the behavior of some malware, we find that they have frequently repeated a specific operation by calling the same API call, like creating a file, sending to a server, or receiving from a server. Some of them also have iterated calling a set of API calls for a long time such as delete

a registry key, create a key, and set a key-value. These cases mostly will be generated and selected as malware signature due to the massive number of their recorded occurrences. Consequently, the chosen features during computing the similarity digest of their signatures will be less than typical cases; this causing either the small value of similarity score ($D_{rate} < 21$) or a small number of selected features ($< N_{min}$) in their signature or the number of bits set to one in $(bf_{sig} \cap bf_{mal})$ is less than the cutoff threshold ($e_{sm} \leq C_t$) and in all cases, the similarity score will be under the detection threshold (t_{score}). So, the score will be recorded as FN.

Our malware detection model achieves 0.974 as the TP and 0.026 as the FN for the VirusShare dataset, whereas we obtain 0.946 as the TP and 0.054 as the FN for the VirusTotal dataset.

However, the FP in both datasets are high (0.386 in VirusShare and 0.306 in VirusTotal) because of several reasons. One of those reasons is the nature of exploiting the API calls by malware. Generally, several events and API calls are frequently used by malware to perform their activities. So, these API calls are vital factors in producing some similarities between different families. Nevertheless, we found that in other cases, there are many differences in the order of using them, which helps the AHBM approach to work appropriately. Moreover, the malware classification process should be implemented based on malware API calls to gain more accurate results instead of depending on antivirus votes election as what the AVClass tool did. Or at least the error rate of misclassification of AVClass should be corrected. This step will improve the results, reduce the FN, increase the TN, and improve malware detection accuracy, precision, recall and F1 score. Furthermore, the number

of malware in each family is crucial in our case. After using the AVClass tool, we found that there are some of the families contain the majority of malware (see tables 3.3 and 3.4). Therefore, the interventions in the behavior of API calls of big families will lead to high FP, so this confirms that there is a need for classifying the datasets based on API calls.

Based on all above results and discussion, we compute the accuracy, precision, recall, and F1 score of our malware detection model depending on the following formulas:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \quad (6.2)$$

$$Precision = \frac{TP}{TP + FP} \quad (6.3)$$

$$Recall = \frac{TP}{TP + FN} \quad (6.4)$$

$$F1 = 2 \times \frac{Recall \times Precision}{Recall + Precision} \quad (6.5)$$

Table 6.8 is shown and grouped the results of our malware detection model. It achieves 0.794 and 0.820 as accuracy, 0.716 and 0.756 as precision, 0.974 and 0.946 as recall, and 0.825 and 0.840 as F1 score for VirusShare and VirusTotal datasets, respectively.

Table 6.8: Results of Malware Detection Using Cross-Validation Evaluation Technique

Datasets	TP	FP	TN	FN	Accuracy	Precision	Recall	F1
VirusShare	0.974	0.386	0.614	0.026	0.794	0.716	0.974	0.825
VirusTotal	0.946	0.306	0.694	0.054	0.820	0.756	0.946	0.840

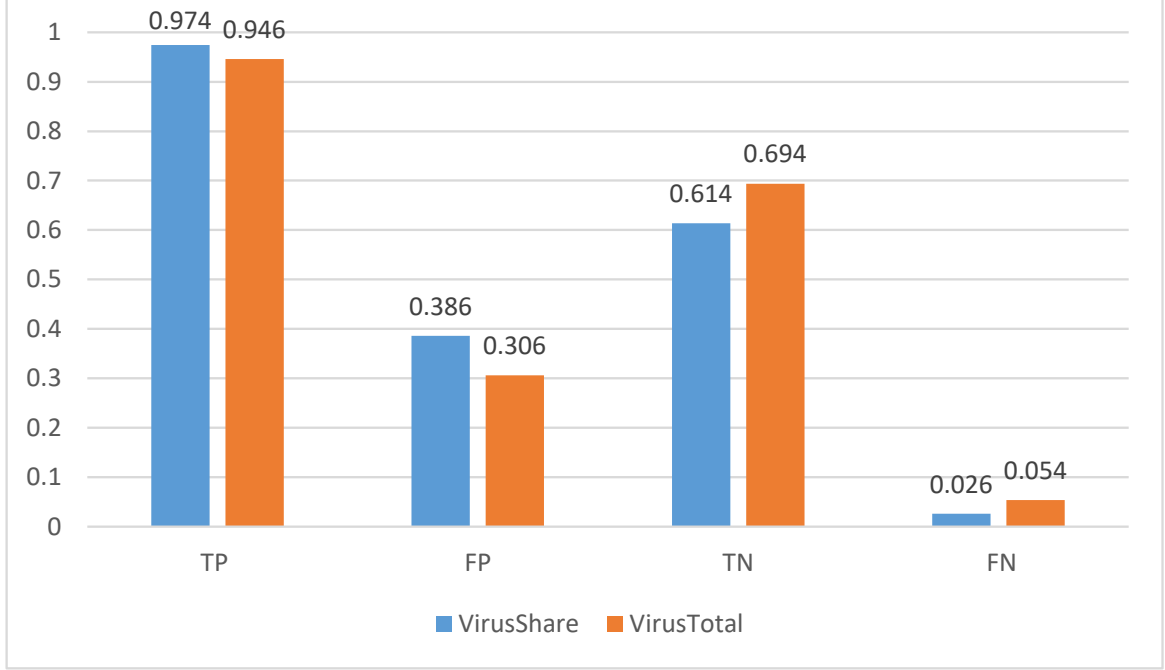


Figure 6.4: Malware Results (TP, FN, FP, and TN)

6.4 Scalability and Time Efficiency

In this section, we aim to test the ability of the AHBM approach to work well with a massive amount of malware API calls and their signatures and measure the execution time for generating the digests and comparing them to detect the malware. For that purpose, we test all training and testing malware samples with all signatures in each fold for both datasets. For example, in the VirusShare dataset, we test all 3176 training samples and all 794 testing samples with all 397 signatures. Consequently, the AHBM approach generates 3176, 794, and 397 similarity digests in around 13.911, 1.981, and 0.813 seconds respectively. Then, it compares the 3176 training samples

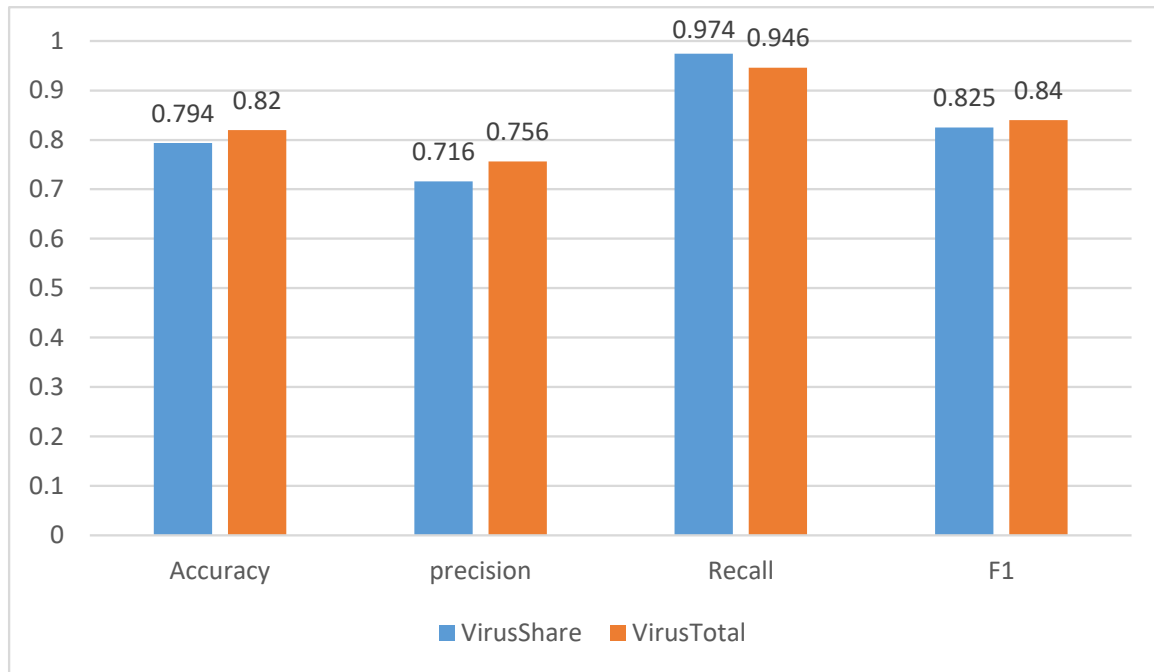


Figure 6.5: Malware Results (Accuracy, Precision, Recall, and F1)

against the 397 signatures in 41.962 seconds and matches the 794 testing samples versus the 397 signatures in 11.385 seconds.

Table 6.9: Results of Scalability and Time Efficiency Experiments for VirusShare Dataset

Phases	Training	Testing	Signatures
No. of Samples	3176	794	397
Digest Generation Time (sec)	3.437	0.862	0.378
No. of Generated Digests in a Second	924.062	921.114	1050.265
No. of detections	1,260,871	315,218	-
Detection Time (sec)	41.962	11.385	-
No. of Detected Malware in a Second	30047.924	27687.132	-

The AHBM shows excellent scalability and works efficiently in time. It matches 1,260,871 cases in around two-thirds of a minute and 7,496,192 in about 4 minutes. Tables 6.9 and 6.10 exhibit the results of scalability and time efficiency experiments. Also, tables 6.11 and 6.12 express the time results of scalability and time efficiency

Table 6.10: Results of Scalability and Time Efficiency Experiments for VirusTotal Dataset

Phases	Training	Testing	Signatures
No. of Samples	7744	1936	968
Digest Generation Time (sec)	13.936	2.037	0.807
No. of Generated Digests in a Second	555.683	950.417	1199.504
No. of detections	7,496,192	1,874,048	-
Detection Time (sec)	245.756	62.092	-
No. of Detected Malware in a Second	30502.58	30181.795	-

experiments for each fold in both datasets.

Table 6.11: Time Results of Scalability and Time Efficiency Experiments for VirusShare Dataset

Time (sec)					
Folds	Digest Generation			Malware Detection	
	Training	Testing	Signatures	Training	Testing
Fold_1	3.4254575	0.8652552	0.3674	40.4163976	12.5246111
Fold_2	3.4870771	0.8577321	0.4229	43.3005193	11.8409707
Fold_3	3.531428	0.8529546	0.3652	41.167079	10.7818168
Fold_4	3.3716064	0.8905603	0.368	42.941632	12.1381662
Fold_5	3.3700162	0.8439463	0.3666	41.985294	9.6392657
Average	3.437	0.862	0.378	41.962	11.385

Table 6.12: Time Results of Scalability and Time Efficiency Experiments for VirusTotal Dataset

Time (Sec)					
Folds	Digest Generation			Detection	
	Training	Testing	Signatures	Training	Testing
Fold_1	13.9112464	1.9810404	0.8129766	229.61322	61.0467422
Fold_2	13.9288175	2.0832417	0.8019621	251.9898383	62.4873113
Fold_3	14.0419386	2.0112937	0.8001932	251.3479287	62.5882879
Fold_4	13.6674519	2.0317722	0.8058674	248.5770358	62.3534181
Fold_5	14.1311764	2.0772895	0.8134386	247.2518328	61.9825573
Average	13.936	2.037	0.807	245.756	62.092

As a summary of this chapter, we show and explain the stages of our malware

detection model. Then, we discuss in detail the procedures of the malware detection process. The cross-validation technique is implemented to evaluate our malware detection model and conduct the scalability and time-efficient experiments. After that, we discuss the findings of our experiments and show the results.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this chapter, we summarize our contribution and show the thesis conclusion. Future work and the limitation of our research are discussed as well.

7.1 Summary of Thesis Contributions

In this thesis, we have started our research by investigating the existing hash-based matching techniques that used typically for digital forensics. We identify the Approximate Hash Based Matching (AHBM) approach as a suitable technique that can be utilized for malware detection based on API calls. This selection depends on the features and capabilities of the AHBM approach against other methods. And to the best of our knowledge, no existing malware detection techniques based on malware API calls use the AHBM approach. So, as a result of this research, we prove the concept of using the AHBM approach for malware detection and present it as a new approach

used for malware detection based on API calls.

ProPeBox environment, one of the main contributions of our research, has been introduced. It is a new protected, controlled, and configurable environment. The ProPeBox environment is protected since it can safely launch the malware and efficiently record their activities with protecting the host machine from malware infection. By using the ProPeBox environment, we have collected two malware datasets from the most popular repositories; VirusShare and VirusTotal. We have got two rich malicious datasets that can be used for multi-purposes of malware researches. ProPeBox has been tested to collect and record malicious activities for more than 19,500 of malware samples.

Besides, we utilize the AHBM approach and adapt it to work on malware API calls. And we use the representative set of characters instead of the real names of API calls to represent all API calls in the same size for malware and signatures sequences. After that, we set up the suitable criteria for the AHBM approach and choosing the proper thresholds values depending on a set of exhaustive experiments.

For malware signature generation, we introduce the simplified method of the Longest Common Substring (LCS) algorithm that optimizes the usage of the LCS algorithm by reducing the execution time of it. We exploit the working nature of the AHBM approach and implement the simplified method of the Longest Common Substring (LCS) by selecting the smallest sample among training samples and extracts the LCS strings between it and others. After that, we implement the filtering and cleaning processes on them to get the output list of sequences. Finally, we set up two criteria for selecting the signatures from the output lists. Based on the experimental

results, Our simplified method of the Longest Common Substring (LCS) proves its effectiveness for generating and extracting the malware signatures sequences.

In this research, the malware detection model contains four stages, which are malware dataset collection, generating malware signatures, generating similarity digest, and malware detection using the AHBM approach. Each step has been shown and explained. Moreover, we have discussed in detail the procedures of the malware detection process. And the cross-validation technique is implemented to evaluate our malware detection model. We partition our datasets into five folds. Each fold has two phases; training (80%) and testing (20%). Indeed, The findings of our malware detection experiments show promising results for using the AHBM approach in the malware detection field.

Furthermore, the results reveal the capability of the AHBM approach to be used for malware detection. The malware detection based on API calls using the Approximate Hash Based Matching (AHBM) approach achieves 0.794, 0.716, 0.974, 0.825 as a result of the accuracy, precision, recall, and F1 score, respectively for the VirusShare dataset. Whereas, for the VirusShare dataset, we have got 0.820, 0.756, 0.946, and 0.840 for the accuracy, precision, recall, and F1 score, respectively. After that, we conduct the scalability and time-efficient experiments. The AHBM shows excellent scalability and works efficiently in time. It matches 1,260,871 cases in around two-thirds of a minute and 7,496,192 in about 4 minutes. In our malware detection model, the findings manifest that more than 30 thousand of malware can be detected in a second.

7.2 Future Work

This thesis has opened the research idea of using the AHBM approach for malware detection. It proves that there is a promising capability for using this approach to detect malware events based on API calls. However, several issues emerge with utilizing the AHBM approach for malware detection. These issues can disclose various ideas to ameliorate this work. The main improvement that can be done for this work is to classify the datasets based on the malware API calls instead of using the AVClass tool. This will lead us to reduce the FP and enhance the precision and recall of the malware detection model.

Also, there is a chance to discover other methods to generate sequences of malware signatures and compare them with our simplified method of LCS. And, the ideas of improvements can cover the AHBM itself. We can also use another type of malicious data from the malware dataset with the API calls to increase the accuracy of our model.

Besides, the ProPeBox can be used to serve multi researches by adding new tools to record -for example- the memory images, network traffic, and other required data for researchers.

REFERENCES

- [1] P. C. Bjelland, K. Franke, and A. Årnes, “Practical use of approximate hash based matching in digital investigations,” *Digital Investigation*, vol. 11, pp. S18–S26, 2014.
- [2] V. Roussev, G. G. Richard III, and L. Marziale, “Multi-resolution similarity hashing,” *digital investigation*, vol. 4, pp. 105–113, 2007.
- [3] J. Kornblum, “Identifying almost identical files using context triggered piecewise hashing,” *Digital investigation*, vol. 3, pp. 91–97, 2006.
- [4] V. Roussev, “An evaluation of forensic similarity hashes,” *digital investigation*, vol. 8, pp. S34–S41, 2011.
- [5] F. Breiteringer, G. Stivaktakis, and H. Baier, “Frash: A framework to test algorithms of similarity hashing,” *Digital Investigation*, vol. 10, pp. S50–S58, 2013.
- [6] V. Roussev, “Data fingerprinting with similarity digests,” in *IFIP International Conference on Digital Forensics*. Springer, 2010, pp. 207–226.
- [7] R. Tahir, “A study on malware and malware detection techniques,” *International Journal of Education and Management Engineering*, vol. 8, no. 2, p. 20, 2018.

- [8] P. Vinod, R. Jaipur, V. Laxmi, and M. Gaur, "Survey on malware detection methods," in *Proceedings of the 3rd Hackers' Workshop on computer and internet security (IITKHACK'09)*, 2009, pp. 74–79.
- [9] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, "A survey on heuristic malware detection techniques," in *Information and Knowledge Technology (IKT), 2013 5th Conference on.* IEEE, 2013, pp. 113–120.
- [10] A. Damodaran, F. Di Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 1–12, 2017.
- [11] K. Mathur and S. Hiranwal, "A survey on techniques in detection and analyzing malware executables," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 4, 2013.
- [12] I. A. Saeed, A. Selamat, and A. M. Abuagoub, "A survey on malware and malware detection systems," *International Journal of Computer Applications*, vol. 67, no. 16, 2013.
- [13] N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, vol. 48, 2007.
- [14] V. Roussev and C. Quates, "Content triage with similarity digests: The m57 case study," *Digital Investigation*, vol. 9, pp. S60–S68, 2012.

- [15] Y. Ye, D. Wang, T. Li, and D. Ye, “Imds: Intelligent malware detection system,” in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2007, pp. 1043–1047.
- [16] Y. Ye, D. Wang, T. Li, D. Ye, and Q. Jiang, “An intelligent pe-malware detection system based on association mining,” *Journal in computer virology*, vol. 4, no. 4, pp. 323–334, 2008.
- [17] M. M. Masud, L. Khan, and B. Thuraisingham, “A scalable multi-level feature extraction technique to detect malicious executables,” *Information Systems Frontiers*, vol. 10, no. 1, pp. 33–45, 2008.
- [18] A. Flaglien, K. Franke, and A. Arnes, “Identifying malware using cross-evidence correlation,” in *IFIP International Conference on Digital Forensics*. Springer, 2011, pp. 169–182.
- [19] J. Bai, J. Wang, and G. Zou, “A malware detection scheme based on mining format information,” *The Scientific World Journal*, vol. 2014, 2014.
- [20] C.-I. Fan, H.-W. Hsiao, C.-H. Chou, and Y.-F. Tseng, “Malware detection systems based on api log data mining,” in *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, vol. 3. IEEE, 2015, pp. 255–260.
- [21] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, “Deep learning for classification of malware system call sequences,” in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2016, pp. 137–149.

- [22] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, “Efficient detection of split personalities in malware.” in *NDSS*. Citeseer, 2010.
- [23] P. Natani and D. Vidyarthi, “Malware detection using api function frequency with ensemble based classifier,” in *International Symposium on Security in Computing and Communication*. Springer, 2013, pp. 378–388.
- [24] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, “A view on current malware behaviors.” in *LEET*, 2009.
- [25] Y. Ki, E. Kim, and H. K. Kim, “A novel approach to detect malware based on api call sequence analysis,” *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, p. 659101, 2015.
- [26] J.-Y. Xu, A. H. Sung, P. Chavez, and S. Mukkamala, “Polymorphic malicious executable scanner by api sequence analysis,” in *Fourth International Conference on Hybrid Intelligent Systems (HIS’04)*. IEEE, 2004, pp. 378–383.
- [27] Y. Qiao, Y. Yang, L. Ji, and J. He, “Analyzing malware by abstracting the frequent itemsets in api call sequences,” in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2013, pp. 265–270.
- [28] M. Alazab, S. Venkataraman, and P. Watters, “Towards understanding malware behaviour by the extraction of api calls,” in *2010 Second Cybercrime and Trustworthy Computing Workshop*. IEEE, 2010, pp. 52–59.

- [29] S. Almuhammadi and O. Bawazeer, “Performance and security tradeoffs in cryptographic hash functions,” *International Journal of Interdisciplinary Telecommunications and Networking (IJITN)*, vol. 12, no. 4, 2020.
- [30] V. Roussev, “Building a better similarity trap with statistically improbable features,” in *System Sciences, 2009. HICSS’09. 42nd Hawaii International Conference on*. IEEE, 2009, pp. 1–10.
- [31] A. Qasem, S. Zhioua, and K. Makhlouf, “Finding a needle in a haystack: The traffic analysis version,” *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 2, pp. 270–290, 2019.
- [32] “virtualbox,” <https://www.virtualbox.org/wiki/Downloads>, accessed: 2019-10-22.
- [33] “procmon,” <https://docs.microsoft.com/en-us/sysinternals/downloads/procmon>, accessed: 2019-10-22.
- [34] “psexec,” <https://docs.microsoft.com/en-us/sysinternals/downloads/psexec>, accessed: 2019-10-22.
- [35] “pandas,” <https://pandas.pydata.org/>, accessed: 2019-10-28.
- [36] “virusshare,” <https://virusshare.com/>, accessed: 2019-10-28.
- [37] “virustotal,” <https://www.virustotal.com/>, accessed: 2019-10-28.

- [38] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, “Avclass: A tool for massive malware labeling,” in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 230–253.
- [39] M. O. Rabin, “Fingerprinting by random polynomials,” *Technical report*, 1981.
- [40] V. Roussev, “Managing terabyte-scale investigations with similarity digests,” in *IFIP International Conference on Digital Forensics*. Springer, 2012, pp. 19–34.
- [41] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.
- [42] A. Broder and M. Mitzenmacher, “Network applications of bloom filters: A survey,” *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.
- [43] “difflib.sequencematcher,” <https://docs.python.org/2/library/difflib.html>, accessed: 2019-11-15.
- [44] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, and S. Mukkamala, “Malware detection using assembly and api call sequences,” *Journal in computer virology*, vol. 7, no. 2, pp. 107–119, 2011.
- [45] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq, “Using spatio-temporal information in api calls with machine learning algorithms for malware detection,” in *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*. ACM, 2009, pp. 55–62.

- [46] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze, “Malware detection based on mining api calls,” in *Proceedings of the 2010 ACM symposium on applied computing*. ACM, 2010, pp. 1020–1025.
- [47] H. Kim, J. Kim, Y. Kim, I. Kim, K. J. Kim, and H. Kim, “Improvement of malware detection and classification using api call sequence alignment and visualization,” *Cluster Computing*, vol. 22, no. 1, pp. 921–929, 2019.

VITAE

- Name: Omar Mohammed Omar Bawazeer
- Nationality: Yemeni
- Place & Date of Birth: Ghail Bawazeer, 02/03/1985
- Email: omar.m.b2010@gmail.com
- Permanent Address: Yemen - Hadhramout - Ghail Bawazeer - Zip code: 1917
- Education:
 - Master of Science in Information Assurance & Security, April 2020, King Fahd University of Petroleum & Minerals, KSA (grade: 3.719 out of 4).
 - Bachelor of Engineering in Computer Engineering, July 2010, Hadramout University of Science & Technology, Hadramout, Yemen.
 - High School Certificate, June 2004, Bawazeer High School, Ghail Bawazeer, Hadramout, Yemen.
- Technical Courses:

- Short course on “Malware Analysis” at ICS department, KFUPM, KSA (22 to 25 February 2017).
 - Short course on “Hands-on Penetration Testing” at ICS department, KFUPM, KSA (15 to 18 February 2017).
 - CCNA course at Modern Technology Institute, Hadhramaut, Yemen (from July to September 2016).
 - Web Development course (ASP.NET with C#) that was a part of Hadramout Academy of College Hires (HACH), at Hadhramaut Institute for Information and Technology, Hadhramaut, Yemen (April to July 2012).
 - Web Development course (PHP) as a summer training at Hadhramaut University of Science & Technology, Hadhramaut, Yemen (July 2010).
- Experiences:
 - Web developer (part-time job) at the Center of Research Excellence for Islamic Banking at Research Institute of KFUPM.
 - Director of the Volunteering Forum of Shabab Ma’ayan El Sheikh, Hadhramaut, Yemen (December 2013 to January 2015).
 - Instructor, Engineering and Petroleum College at Hadhramaut University of Science & Technology, Department of Computer Engineering, Hadhramaut, Yemen (September 2011 to January 2015).
 - Instructor (part-time work), University of Science & Technology, Department of I.T., Hadhramaut, Yemen (March 2013 to January 2015).

- Instructor (part-time work) Al-Rayan University, Department of I.T., Hadhramaut, Yemen (September 2011 to March 2013).
 - Instructor, Al-Rayan University, Department of I.T., Hadhramaut, Yemen (October 2010 to August 2011).
- Published Researches:
 - “Performance and Security Tradeoffs in Cryptographic Hash Functions” published in International Journal of Interdisciplinary Telecommunications and Networking (IJITN), Volume 12, Issue 4, 2020.
 - “Top NewSQL Databases and Features Classification”, published in International Journal of Database Management Systems (IJDMS), volume 10, 2018.
- Programming:
 - Languages: C, C++, Java, C#, and Python.
 - Databases: MYSQL, and SQL server.
 - Web development: PHP, ASP.NET, HTML, XHTML, CSS2, CSS3, JavaScript, jQuery, and AJAX.
- Languages:
 - Arabic (mother language).
 - English (fluent).