Behaviour-based Virus Analysis and Detection

PhD Thesis

Sulaiman Amro Al amro

This thesis is submitted in partial fulfilment of the requirements

for the degree of Doctor of Philosophy

Software Technology Research Laboratory

Faculty of Technology

De Montfort University

May 2013

DEDICATION

To my beloved parents

This thesis is dedicated to my **Father** who has been my supportive, motivated, inspired guide throughout my life, and who has spent every minute of his life teaching and guiding me and my brothers and sisters how to live and be successful.

To my **Mother** for her support and endless love, daily prayers, and for her encouragement and everything she has sacrificed for us.

To my **Sisters** and **Brothers** for their support, prayers and encouragements throughout my entire life.

To my beloved Family, My **Wife** for her support and patience throughout my PhD, and my little boy **Amro** who has changed my life and relieves my tiredness and stress every single day.

ABSTRACT

Every day, the growing number of viruses causes major damage to computer systems, which many antivirus products have been developed to protect. Regrettably, existing antivirus products do not provide a full solution to the problems associated with viruses. One of the main reasons for this is that these products typically use signature-based detection, so that the rapid growth in the number of viruses means that many signatures have to be added to their signature databases each day. These signatures then have to be stored in the computer system, where they consume increasing memory space. Moreover, the large database will also affect the speed of searching for signatures, and, hence, affect the performance of the system. As the number of viruses continues to grow, ever more space will be needed in the future. There is thus an urgent need for a novel and robust detection technique. One of the most encouraging recent developments in virus research is the use of formulae, which provides alternatives to classic virus detection methods. The proposed research uses temporal logic and behaviour-based detection to detect viruses. Interval Temporal Logic (ITL) will be used to generate virus specifications, properties and formulae based on the analysis of the behaviour of computer viruses, in order to detect them. Tempura, which is the executable subset of ITL, will be used to check whether a good or bad behaviour occurs with the help of ITL description and system traces. The process will also use AnaTempura, an integrated workbench tool for ITL that supports

our system specifications. AnaTempura will offer validation and verification of the ITL specifications and provide runtime testing of these specifications.

DECLARATION

I declare that the work described in this thesis is original work undertaken by me for the degree of Doctor of Philosophy, at the Software Technology Research Laboratory (STRL), at De Montfort University, United Kingdom.

No part of the material described in this thesis has been submitted for any award of any other degree or qualification in this or any other university or college of advanced education.

I also declare that part of this thesis has been published in some of my following publications.

Sulaiman Amro Al amro

PUBLICATIONS

1. S. Al Amro., Aldrawiesh, K. and Al-Ajlan, A. A Comparative study of Computational Intelligence in Computer Security and Forensics*. The 2011 IJCAI Workshop on Intelligent Security (SecArt), 2-9*. Barcelona, Spain: AAAI Press., 2011.

2. S. Al Amro, Cau, A. *Behaviour-based Virus Detection System using Interval Temporal Logic*. The 6th International Conference on Risks and Security of Internet and Systems (CRISIS 2011), 2-9. Timisoara, Romania: IEEE Computer Society., 2011.

3. S. Al Amro, F. Chiclana, D. A. Elizondo*. Application of Fuzzy Logic in Computer Security and Forensics.* In: Computational Intelligence for Privacy and Security. David Elizondo, Agusti Solanas, Antoni Martinez-Balleste (editors), Springer Series: Studies in Computational Intelligence., 2012.

4. S. Al Amro, F, A. Elizondo, A. Solanas, and A. Martínez-Balleste: *Evolutionary Computation in Computer Security and Forensics: an Overview.* In: Computational Intelligence for Privacy and Security. David Elizondo, Agusti Solanas, Antoni Martinez-Balleste (editors), Springer Series: Studies in Computational Intelligence., 2012.

5. S. Al Amro and Cau, A. *Behavioural API based Virus Analysis and Detection*. International Journal of Computer Science and Information Security, 10(5):14–22, May 2012.

ACKNOWLEDGMENTS

First of all, all thanks and praise would first go to **God (Allah)** for all the success.

My sincere thanks would go to my supervisor **Dr. Antonio Cau** for all his support, time and guidance. This thesis would not have been completed without the in-depth discussions and comments from **Dr. Antonio**. I also would like to thank my second supervisor **Dr. Giampaolo Bella** and **Prof. Hussein Zedan,** the head of the STRL, for their insightful comments and advice.

My many thanks would go to the Cyber Security Centre (CSC) at De Montfort University for letting me use their Forensics and Security Laboratory to do my analysis, experiments and testing. My special thanks go to **Mr. Gareth Lapworth,** who is the expert on computer viruses, for his hours and hours of discussions and his help in understanding computer virus analysis tools.

My special thanks go to the developers of Deviare API tool for letting me use their source code to develop my research, especially **Mr. Mauro Leggieri** for the patience with which he checked and corrected many technical errors.

I also would like to thank the other STRL staff who have given me some of their precious time to comment on my work. A special thanks to my assessor **Dr. Francois Siewe** for his guidance and comments. I also would like to thank all my colleagues at the STRL for the valuable advice and discussions and a special thanks to my office mate **Mr. Fahad Alqahtani.**

My sincere thanks would go to all my family (my parents, my wife, my sisters and brothers) for their support and prayers.

Leicester, England, 2013

Sulaiman Al amro

Table of Contents

List of Figures

List of Figures

Listings

Listings

List of Tables

List of Acronyms

AV	Antivirus
BSM	Basic Security Module
EICAR	European Institute for Computer Antivirus Research
ITL	Interval Temporal Logic
LTL	Linear temporal logic
CTL	Computation Three Logic
Malware	Malicious software
FP	False Positive
FN	False Negative
LAN	Local Area Network
NAA	Network Application Architecture
ACN	Abstract Communication Network
VM	Virtual Machine
OS	Operating System.
PE	Portable Executable.
OEP	Original Entry Point
API	Application Program Interface.
SSDT	System Service Dispatch Table.
IAT	Import Address Table.
PID	Process Identifier
GSR	Gene of Self-Replication
DCFS	Document Class wise Frequency Feature Selection

PA	Propositional Variables
CTPL	Computation Tree Predicate Logic
C#	C-Sharp

Chapter 1

Introduction

Objectives:

- Provide an overview of the motivations and research problems.
- Highlight the research hypotheses and success criteria.
- Identify the scope of research.
- Describe the research methodology.
- Present the thesis structure.

1.1 Preface

The rapid growth of technology and the speed of communication that the internet provides make research into computer viruses detection essential, in order to protect computers and network systems from any malevolent attack. It is highly recommended that virus researchers should be aware of new trends, which virus writers will exploit whenever they have the opportunity. The success that attackers enjoy demonstrates that there needs to be a novel and robust detection system to prevent attacks. Since they first appeared, computer viruses have caused disruption to private and public organisations, governments and computer users, as they attempt to remove, modify or steal sensitive data. Therefore, a robust system is needed in order to minimise these disturbances and to defeat the new techniques used by skilful attackers.

Traditional antivirus (AV) products provide a good detection technique which relies on signatures that have been collected from previous known viruses and then added to an AV database. Prior to the arriving of a virus to the system, its signature will be compared with those stored in the database and if there is a match, the virus will be detected; otherwise, the system will run normally [1]. Thus, newly released viruses will not be detected by signature-based detection unless the antivirus company receives this new virus and stores its signature in its own database. The other disadvantage of this approach is that it needs a large database in order to store the signatures. As the number of viruses increases every day, ever larger databases are needed to store all their signatures, so that more storage space will be needed in the near future [2, 3]. The large database will also affect the speed of searching for signatures, and, thus,

affect the performance of the system. These disadvantages mean that the signature-based detection technique will soon be inadequate to protect computer systems.

Another and more promising approach which has been applied recently is called behaviour-based virus detection. As their name suggests, such techniques do not rely on a database of signatures, but instead concentrate on the behaviour of the system. Behaviour-based virus detection has come to light in order to overcome the problems associated with traditional signature-based detection. The principle behind this approach is first to observe the normal behaviour of the system, after which any deviation from it will be classified as an intrusion [4]. The second arm of behaviour-based detection is to predefine virus behaviour, so that any process which resembles virus activity can be identified as a potential virus. However, there are difficulties associated with behaviour-based detection, the greatest of which is how to define the behaviours that will represent known and novel viruses without confusing them with normal processes running in the system (known as false positives) [5]. In addition, some existing virus behaviour detection techniques rely on detecting subclasses of viruses. In general, behaviour-based detection techniques rely on identifying virus characteristics in order to detect these viruses and other viruses sharing the same characteristics in the future. One of the objectives of this research is to look deeply into the characteristics of computer viruses in order to deduce properties, and specifications that will be used in this research.

The problem of the ever growing number of computer viruses leads virus researchers to seek alternative solutions. The representation of virus behaviour using

formulae is one of the most promising techniques used to detect computer attacks nowadays [6]. The present research proposes to build a detection technique using temporal logic specifications that have been inferred from the analysis of virus behaviour. We believe that using such logical specifications and formulae will minimise the problem of the rapidly growing database of traditional AV products. The goal of the detection technique to be used in this research is to detect viruses with a low consumption of memory space, so that the system will run at an acceptable speed even if many novel computer viruses are released. A logic called Interval Temporal Logic (ITL) has been chosen to be used in this research because this logic is very suitable to describe system traces, i.e., it can be used to describe bad and good behaviours. The Tempura tool will be used to check whether a good or bad behaviour occurs with help of ITL description and system traces.

1.2 Motivation

One of the greatest challenges facing computer researchers today is the rapidly growing number of computer viruses, which obliges signature databases to become ever larger. According to Kaspersky [7], more than 36,000 new viruses appear every day, and these all need to be analysed in order for their signatures to be added to the signature database. This is a lengthy process and requires a large amount of storage space. Furthermore, Norton [8], states that computers are always threatened by malwares due to the thousands of new viruses, worms and Trojans which are created every day. As new viruses arrive, AV databases are becoming ever larger, making systems run increasingly slowly as available space is reduced [9]. This concern led to

the decision to examine a selected antivirus product (Kaspersky) in order to discover exactly how much system memory space these signatures require. On 14th March 2011 it was observed that there were 5495548 signatures in the Kaspersky AV database, as shown in Figure 1.1. According to Song [10], the average length of a virus signature is 67 bytes. Therefore, the following calculation shows that the size of the database in question would be over 350 MB:

5495548 (signatures) * 67 bytes = 368201716 bytes ≈ 351.2 megabytes

This is means that to store these signatures, the system would need a large memory; and as the number of viruses grows every day, increasing space would be needed for the AV database. Ten days after the above observation, i.e., on 24th March 2011, the number of signatures in the same AV database was observed to have increased from 5495548 to 5557346. This set of observations demonstrates that it is essential to deal with the daily growth in the number of viruses.

Figure 1.1: The number of signatures in a selected antivirus product.

1.3 Research Problems

Research reports [2, 3, 9] argue that signature-based virus detection will not be sufficient in the near future because of the continual increase in virus numbers, the increased memory space required and the consequent reduction in performance and speed of the systems concerned. In addition, the large database will also affect the speed of searching for signatures, and, thus, affect the performance of the system. Other studies [6, 11] argue that there is a growing need for formulae to be used in detecting attacks, providing a robust and manageable detection technique. This debate in the field of computer security raises a number of questions and problems that will be addressed by the proposed research. **The central research problem is to find ways of detecting computer viruses by their behaviours without consuming large memory space**. A formal mechanism will be adopted in order to investigate this problem. Currently, there is little research into behaviour-based detection techniques which use logical formulae to detect viruses in the real world. The second problem addressed by this research is to assess **the effectiveness of behaviour-based virus detection using logical formulae, its main purpose being to minimise the use of memory space with respect to the number of false positives and false negatives.**

1.4 Research Hypotheses

Research hypotheses are very important in the preparation and development of research due to the fact that they can be used as a guide for the research process. In addition, they can help with determining the types of data that are needed to

investigate this research. "*A hypothesis is a logical supposition, a reasonable guess, an educated conjecture. It provides a tentative explanation for a phenomenon under investigation*" [12]. According to Burns [13], normally the process of supporting or rejecting the hypothesis itself is not tested but in fact the so-called *null hypothesis,* which is the opposite or negation of the hypothesis, is the aspect which is tested. As a result, if the null hypothesis is inadmissible then the original hypothesis is admissible. In addition, the original hypothesis can be called the *alternative hypothesis,* that is the alternative to the null hypothesis. As a result, two hypotheses are expressed namely, the alternative hypothesis that is represented by the symbol *H1* and which is the original one and the null hypothesis that is represented by the symbol $H1_0$ which needs to be rejected.

The research hypotheses, which include the alternative and null hypotheses, will be formulated according to the research problems aforementioned. Therefore, two hypotheses will be formulated and the following is the alternative hypothesis and its null hypothesis of the first problem:

The first hypothesis:

H1: Detecting computer viruses by their behaviours can obtain results without consuming large hard drive space.

The first null hypothesis:

$H1_0$: Detecting computer viruses by their behaviours can **NOT** obtain results without consuming large hard drive space.

The alternative hypothesis and its null hypothesis of the second problem will also be formulated as follows:

The second hypothesis:

H2: The number of false positives and false negatives when detecting known and unknown computer viruses by their behaviours using AnaTempura **is the same as** or **lower than** the number when detecting known and unknown computer viruses by their signatures.

The second null hypothesis:

$H2_0$: The number of false positives and false negatives when detecting known and unknown computer viruses by their behaviours using AnaTempura is **higher than** the number when detecting known and unknown computer viruses by their signatures.

1.4.1 Hypothesis Testing

To test (answer) the research hypothesis, statistics are normally used. According to Mark and Ronald [14], statistics are usually classified as either descriptive or inferential:

- Descriptive statistics are used to describe, show or summarize data in a meaningful way. It can provide information about a group of data. Descriptive statistics might include graphical and/or numerical techniques for showing concise summaries of data [13]. Therefore, descriptive statistics cannot reach conclusions regarding any hypotheses that have been made.

- Inferential statistics use samples to make generalisations about the populations from which the samples were drawn, based on information obtained from that sample. That is to say, Inferential statistics can be used to provide a valid conclusion about the population using the descriptive statistics. As a result, inferential statistics are used to answer our research hypotheses [13].

1.5 Success Criteria

The following points indicate the success criteria of this research. These success criteria have been formulated In order to measure the success of this research.

- Detecting known and unknown viruses (viruses that have not been analysed before).

- False positive production.

- Running with an acceptable system performance.

- The ability for users to interact with the operating system while the prototype is running.

1.6 Scope of Research

There are many different kinds of malicious software (malware) which infect the system in different ways, and thus cannot be detected by a single detection system [1, 15, 16]. As the title of this thesis suggests, this approach will only concentrate on detecting one specific kind of malware which is computer viruses. The reason why this approach is only targeting one type of malware is because we believe that it is better to concentrate on a single detection system first and then improve it in the future to target other types of malware. Hence, the scope of this research includes in particular:

- **Computer Virus Attachment**

Due to the fact that a malware must attach itself to another file or another executable program in order to be called a computer virus [17], this research will only concentrate on those which infect other files or programs. Therefore, any virus that follows the theory of attachment and infect another file within the operating system, such malware would be targeted and detected by the present approach. The infection time of a virus should not be large (that is, a few seconds or minutes maximum) in order to be detected by our prototype. In addition, we assume that AnaTempura is not infected during the runtime testing.

- **Window 32 Applications**

According to Boyce [18], Windows 32 applications (including normal processes and computer viruses) can run in both 32-bit and 64-bit operating system, however, a 64-bit application will only run on a 64-bit operating system. As a result, this research will

only concentrate on, analyse and examine 32-bit normal processes and computer viruses.

1.7 Research Methodology

The scientific research is the research method that is used in the present approach [19]. The constructive research approach has been chosen and is followed by the adopted research. Contribution to knowledge with the constructive method means that a new solution will be developed for the identified problem [19]. In this research, a new framework has been developed for known problems which are consuming system memory space and detecting unknown viruses.

The scientific research methodology in this research is conducted by the following work packages. Firstly, research background and literature review. The second work package deals with the suggested framework on which this research is based with. The third work package explains in detail the implementation of the proposed framework. The final one evaluates the research findings and results.

- **Research Background**

  The research study starts by reviewing the literature in the area of computer viruses. Firstly, the differences between computer viruses and other types of malware are defined. Then, traditional and behaviour-based virus detection techniques are distinguished. Finally, a critical review of published works in the area of behaviour-based virus detection using various techniques are provided.

In addition, a comprehensive study of previous work, comparing it with the proposed framework, assists us in identifying its drawbacks and limitations.

- **Architecture**

The design of the framework is concentrated in the architecture work page. All the components of the proposed framework are specified in this work package. The interaction between these components in achieving the research aims are identified in this work package. The process involved in the work package is divided into two tasks:

1. Virus Analysis.
2. Virus detection.

- **Prototype Implementation**

In this work package, the design and implementation of the prototype are described. The prototype of this study implements the detection architecture that leads to a runtime monitoring system for computer viruses. In this work package, a real implementation which interacts with different Microsoft Windows operating systems has been built. The information which comes from these operating systems is delivered to Tempura (The executable subset of ITL) which examines them for any interesting behaviour.

- **Evaluation**

The capability of the framework and its components will be evaluated in this work package. The research hypotheses are tested in this work package to

examine the effectiveness of the proposed research. Whether the proposed research has successfully detected known and unknown computer viruses without memory consumption will also be examined in this work package.

1.8 Ethical Principles

When dealing with viruses in a network, they might go through this network and cause damage across the entire organisation. Alternatively, a virus might be received by a person inexpert in security, who might panic for several reasons. Therefore, a secure environment is needed in order to conduct research into computer viruses. According to Zeltser [20], in order to build a malicious software (malware) analysis environment, the first two steps are to use a virtual machine for the laboratory analysis, so that no actual machine is harmed, and to isolate this strongly from the production environment in order that the viruses cannot spread.

According to the European Institute for Computer Antivirus Research (EICAR) [21], computer viruses and malware codes are considered to pose a threat to computer users, developers and virus researchers alike. The code of conduct of which EICAR requires its members to be aware has three elements:

3. Total abstinence from activities or publications which could cause or foster panic, i.e., no "*trading on people's fears*".
4. Abstaining from unwarranted superlatives and factually untenable statements in advertising, e.g. "All known and unknown viruses will be recognised".

5. Information which may assist the development of viruses or other malicious program code will not be published or given to a third party, with the exception of the exchange of such information with institutions, companies and persons responsibly researching in this sector or active in combating malware.

The EICAR code of conduct binds its members, most of whom are AV researchers. The present researcher will therefore follow the code in order not to cause concern to other people. Specifically, I make the following declaration:

1. I will not intentionally cause harm to live data and will not change any data unless authorised to do so.
2. I will not pass or exchange any viruses or malicious codes to other people who are not involved in my research.
3. I will follow at all times the code of conduct which EICAR imposes on its members and will behave accordingly.

1.9 Thesis Outline

Including this chapter, this thesis is organised in eight chapters. The following briefly summarises each one:

Chapter Two: Literature review

This chapter provides an introduction to the principles of computer viruses. It explains the various types of virus detection systems. It then concentrates more on behaviour-based detection and the published works related to the present research. The system

service that is found as the best to extract virus behaviours, which is Application Programming Interface (API) call, is also addressed in this chapter.

Chapter Three: Preliminaries

This chapter provides a brief explanation of the proposed framework. The syntax and semantics of Interval Temporal Logic (ITL) which is the base language of Tempura are detailed. The reasons for the choice of ITL are addressed in this chapter. In addition, related works which use similar tools will be discussed and then criticised.

Chapter Four: Framework for Behavioural Detection of Viruses

This chapter proposes the main framework, which is the base of the two architectures, with which this research will work. These two architectures are namely, virus analysis architecture and virus detection architecture.

Chapter Five: AnaTempura Integration and Implementation

This chapter will present the prototype of the research, using Tempura, including the kernel rootkit and Deviare API tools which help to deliver the Native and Win32 API information to Tempura that is used to examine them.

Chapter Six: Case Studies

This chapter outlines the number of investigations which are carried out to examine the theory validation of this research.

Chapter Seven: Results: Analysis and Evaluation

The test results chapter provides the results for both the virus analysis and the

prototype. It also evaluates the proposed research and discusses its limitations.

Chapter Eight: Conclusion and Future work

This chapter summarises the research and recommends a new direction for future work. It provides the thesis summary, contributions, revisited success criteria, limitations and future work.

Chapter 2

Literature Review

Objectives:

- Provide a background about computer malware, specifically, computer viruses.
- Provide a background about computer virus detection techniques.
- Discuss in detail the behaviour-based virus detection.
- Highlight the system service that is used to detect virus behaviour.
- Identify the gaps of the related research.

2.1 Introduction

This chapter provides an overview of computer viruses. It starts with a background about computer viruses that tries to answer a number of important questions which should be asked about computer viruses, such as what they are, how they are able to spread and harm individual computers, who writes them, what they wish to accomplish and what techniques have been used to defend our systems from viruses. Then, definitions of some terms that will be significant in this research will be provided. After that, more details about computer viruses and the famous types of computer viruses will be discussed. Section 2.5 will discuss different techniques used to detect computer viruses besides their advantages and drawbacks. Moreover, Section 2.6 will concentrate on behaviour based virus detection which is the technique used in this research. Subsequently, Application Programming Interface (API) that is known to be one of the best system services to trace virus behaviour [22, 23] will be introduced in this chapter. At last, motivations and related works to our research will be discussed and criticised.

2.2 Background

Since the late 1980s, when the first serious computer virus appeared, a war has been waged between virus generators and the antivirus community [5]. This struggle continues to this day, thanks to the daily discovery of new techniques for generating viruses and defending systems against them. In April 2006, Kaspersky [24] reported that every month, there were over 10 thousand updates to a particular antivirus (AV)

program in response to the discovery of new viruses. Early in the same year, the FBI calculated the cost of computer crime for several companies and reported that computer viruses caused the greatest losses (67% of overall losses) [25]. Indeed, it is apparent that computer viruses have a high cost for both individuals and organisations. Therefore, dealing with them is an essential, never-ending task and research in the antivirus field is required in order to minimise the associated threats and losses.

2.3 Taxonomy of Malicious Software

According to Davis [5], there are a number of terms in the field of computer viruses which might be confused, such as Trojan horse, backdoor, worm and malicious software. The following paragraphs offer definitions of these types of malware and explanations of the terminology [5, 26].

Malware is an abbreviation of the phrase 'malicious software', which can be defined as a computer program which attempts to harm the system without the knowledge of the computer user. There are several categories of malware, including worms, viruses, Trojan horses, backdoors, bombs and rootkits [1].

A Trojan horse is a program that appears to be legal and which once executed gives the attacker unauthorised remote access to a system or can be extended to download more malicious software.

A virus is a code that recursively replicates a possibly evolved a copy of itself. In other words, it is a computer program that attaches itself to other files or processes.

A worm is a program that is designed to infect host machines by individually replicating itself across networks.

Rootkits are special tools used by an attacker after breaking into a computer system, in order to obtain root-level access.

A backdoor is a program that attempts to bypass the defence system in order to gain unauthorised access to a computer.

In addition to these definitions there is another classification of malware which includes programs like adware and spyware that are not dangerous in themselves but still harm the system by reducing its performance, exposing new vulnerabilities and weakening it in ways that might affect its usability. This malware is called **grayware** [5].

Identification is not always accurate, so the performance of a computer security system should be considered in terms of the extent of false positives and false negatives. **False positives** occur when normal (benign) programs are identified by a defender as malicious, while **false negatives** are when malicious programs are not detected but rather classified as normal.

There are many subcategories of malware other than the five most common ones, which have been defined here. In order to give a clear definition and to distinguish them clearly from other malicious software, the next subsection offers a comprehensive examination of computer viruses.

2.4 Computer Viruses

Since their emergence in the 1980s, a large number of definitions have been put forward by many researchers as to what constitutes a computer virus. Fred Cohen, who invented the technique of defence against computer viruses, defined a virus as a program that can 'infect' other programs by modifying them to include a (possibly evolved) version of itself [16]. Later, in 2005, Peter Szor claimed that the former definition is incomplete because it does not incorporate all viruses. He defines a computer virus as "A program that recursively and explicitly copies a possibly evolved version of itself" [1].

A virus must attach itself to other programs because it is unable to be executed by itself and that is one of its main characteristics [27]. Computer viruses have succeeded in satisfying the desires of their writers, especially in spreading, causing damage and bypassing detection. Nowadays, because computers have become very important to individuals, governments and organisations, computer viruses constitute a major problem of daily life and it is one of the fundamental aspects of computing that people should be aware of [1, 4].

One of the earliest papers on computer viruses was written by Cohen and Adleman [15, 28]. Cohen was the first to use the term 'virus' and using Turing machines, he also proposed the first formal definition of computer viruses [15]. Cohen states that the only way to be fully protected against viruses is by isolating the system, but notes that this cannot be practically implemented. He concludes that for a system

to be secured against viruses, it must be protected from interference with both outgoing and incoming information flows.

Adleman built upon Cohen's work by inventing more formal definitions and classifications of computer viruses. His conclusion was that any program which has been infected will cause one of the three following types of damage [28]: first, impairment of the system by doing an injury to it; second, harm to the system by replicating itself in other programs; finally, producing an imitation of itself when it cannot find a file to infect.

Computer viruses have been classified as simple and complex [5]. Simple viruses have been the backbone of malicious software for the last 25 years and can be divided into three types: file viruses, boot sector viruses and macro viruses. Within these groups, a wide range of strategies are used by virus writers in order to infect files [1, 4].

Overwriting viruses: In this method, the target code will be removed by the virus and an infected file is replaced with it.

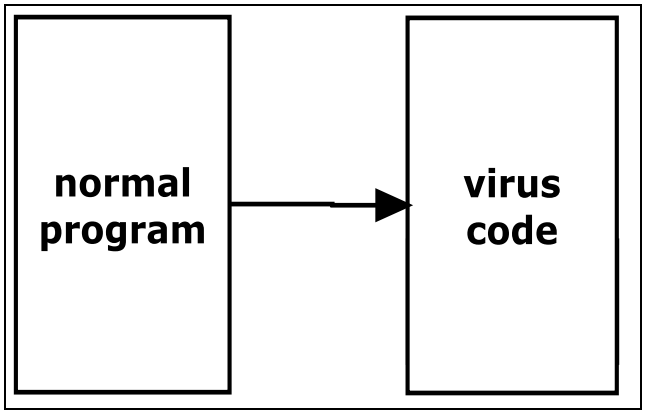


Figure 2.1: Overwriting virus.

Parasitic viruses: Here, a virus code will be inserted into the existing file to gain control of it. Parasitic viruses include appending and prepending types.

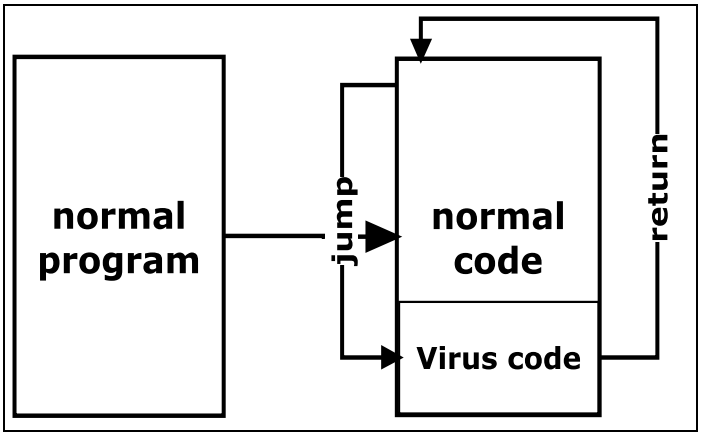


Figure 2.2: Appending virus.

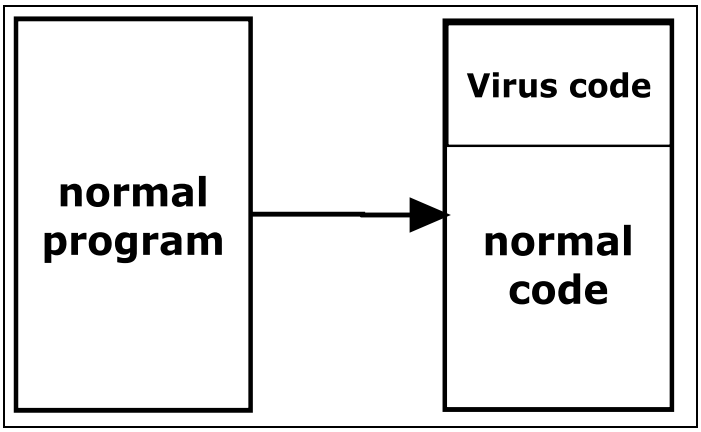


Figure 2.3: Prepending virus.

Companion viruses: The target file will be duplicated by a companion giving a copy of the original file that contains the virus in it.

Link viruses: A link to the virus file will be incorporated into the target file.

Application source code viruses: An active virus can be included in the source code of some applications during their installation.

Cavity viruses: These are viruses that do not increase the size of the infected file, but instead overwrite part of it by including the virus code.

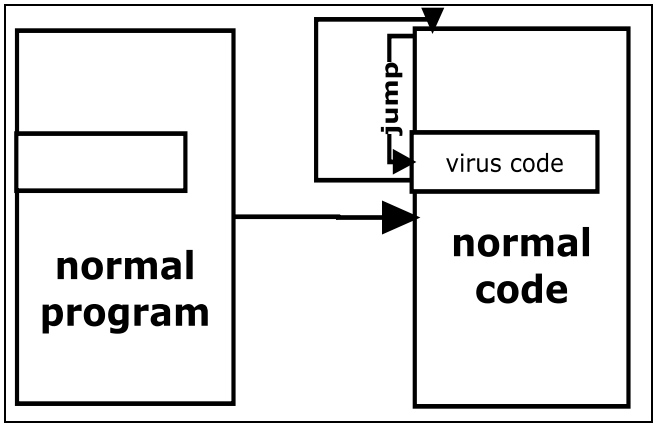


Figure 2.4: Cavity virus.

Compressing viruses: As their name suggests, these viruses compress the content of the host program. The purpose of this technique is to hide the increase in the file's size after an infection has occurred.

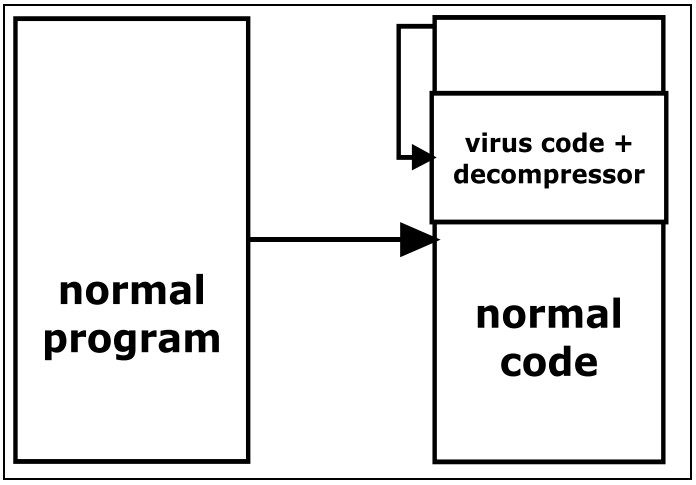


Figure 2.5: Compressing virus.

The master boot record, which is a type of boot sector, is normally infected with what is called a boot sector virus. The final simple virus type is the macro virus or shortcut virus, which normally repeats itself. Despite the fact that the use of these

macros can be very helpful, they can also cause great damage to the system [5]. Macros can be loaded automatically when Microsoft Office applications are loaded. Therefore, the virus have an excellent opportunity to launch without notifying the user. For example, a user might receive an email contacting an attached Microsoft Word document. When the attached file is opened, the Word document launches and the macro virus is loaded on the target system.

On the other hand, there are the complex or advanced viruses which have been invented by virus writers in order to evade detection techniques [1, 5]. With the evolution of defence techniques, virus writers are forced to invent viruses that are difficult for antivirus systems to detect. These can be classified into the following subcategories [1, 5]:

Encrypted viruses are encrypted in order to avoid antivirus software. This type comprises the first attempt to generate a complex virus. It was a successful technique to avoid the old signature-based detection techniques (to be discussed in Section 2.5.1.1).

Oligomorphic viruses: This is the next decrypting technique which is normally detected by AV programs, where the decryption technique is randomly generated. It differs from basic encryption by having a set of decryptors rather than only one.

Polymorphic viruses: This is the most common decrypting technique ever used. The idea is that it can change its decryptors, which can take an unlimited number of forms. Polymorphic viruses have proved to be the hardest type for antivirus programs to detect.

Metamorphic viruses differ from the others in not having a decryptor; rather, they have the ability to construct new generations that look different. The significant feature of metamorphic viruses is that they do not change the whole code, but only its functionality.

Entry-point obscuring viruses: The idea of this technique is that a code is randomly written to a location within an existing program and appears to give an update to this program. The trick is that when the trusted program is executed, the system automatically executes the virus code.

A virus can spread from program to program in the same system and can also be transferred from user to user via a network [16]. With the rapid evolution and improvement of the Internet, there have been various ways in which a virus can spread and infect systems [5], one of the best known being by email. This happens when a file is attached to a message in the mailbox; once a user clicks to open this attachment, the virus spreads. In addition to emails, downloads from the Internet, especially malicious websites, are important in spreading viruses. Removable media such as floppy disks, CDs and USBs can also cause great damage by carrying viruses in them. Users should be aware of these methods.

Computer viruses can cause low to very high damage to a system, including the removal of all information on the hard drive [1, 16, 27]. A common type of virus damage is denial of service, where the computer's resources are kept so busy that the system is unavailable to the user. Some viruses are constructed to damage certain hardware by removing all information from it (formatting), overwriting it or even

destroying it. Another risk is the stealing of data from a system. Some virus writers make money by accessing individual's systems and stealing their credit card numbers and other important information in phishing attacks, using backdoor features, for example [1]. However, this research will only concentrate on those which infect other files or programs. Therefore, any virus that follows the theory of attachment and infect another file within the operating system, such malware would be targeted and detected by the present approach.

2.5 Computer Virus Detection

In the late 1980s and early 1990s, writing antivirus software was not very hard because at that time many individuals could create one. Two papers [15, 28] opened a path for computer virus researchers to establish a number of studies in the field of virus detection. Despite this, antivirus techniques have been developed successfully in dealing with computer viruses during the last 25 years. Virus detection techniques can be defined according to how the presence of a virus can be identified in an object [29].

A great number of detection techniques have been discussed [30], with their advantages and disadvantages. However, there are two basic detection techniques which can be distinguished, namely manual (on-demand scanning) and on-access (real-time scanning) [1, 5].

- On-demand scanning is a simple virus detection technique where the user initiates the scan. This technique is not sufficient to deal with dynamic malwares such as macro viruses. In addition, it is an offline scan that cannot

detect a virus unless the user is aware of it and allows scanning; otherwise, the virus will infect the system. Most AV products use this type of detection as a secondary capability [5].

- The other type of scanning, which is called on-access, dynamic or real-time scanning, is a more powerful technique because of its ability to detect more complex viruses [4]. This type of detection normally happens without the knowledge of the user. The AV product scans the system memory and the hard disk looking for viruses, as the computer user browses email, opens an application or downloads cyber-content. In this technique, if a virus is detected the malicious activity will be halted, then the user will be notified and advised to take action. This type of detection is commonly used in the commercial market today.

2.5.1 On-access Scanning

There are two types of on-access scanning: signature-based detection and anomaly or heuristic-based detection [1, 5].

2.5.1.1 Signature-based Detection

Signature-based detection works by searching for particular sequences of bytes within an object in order to identify exceptionally a particular version of a virus [29]. Also known as string scanning, it is the simplest form of scanning, constructed upon databases which have virus signatures. When a new virus emerges, its binary form will be specifically and uniquely analysed by a virus researcher and its sequences of bytes

will be added to the virus database [29]. A virus is identified by its sequences of bytes and what is called a virus signature. In addition, a hash value is another type of signatures. A large amount of data is converted into a single value by a mathematical function or a procedure known as a hash function [31].

Most AV products around the world use the signature-based technique and are trying to develop it, despite the fact that it is not sufficient for most viruses (as will be discussed later). Indeed, it has certain limitations that make it not good enough to meet the evolution and acceleration of new technologies [5]. One of its greatest weaknesses is that it is based on signature databases, which need to be updated regularly. Therefore, two actions are required: a list of signatures must be produced by the vendor, then downloaded and installed by the consumer.

Another important drawback of this approach is that it needs a large database in order to store the signatures. As the number of viruses increases every day, ever larger databases are needed to store all their signatures, so that large storage space will be needed in the near future. The large database will also affect the speed of searching for signatures, and, thus, affect the performance of the system. These disadvantages mean that the signature-based detection techniques will soon be inadequate to protect computer systems [32].

In addition, many viruses today can mutate in various ways, including polymorphic and metamorphic viruses. Because signature-based detection can only identify and detect the signatures in its databases, these viruses will normally defeat the engine and bypass the defender. One of the important capabilities that signature-

based detection lacks is the detection of unknown and novel viruses. For each new virus to be discovered and added to the consumer update list, antivirus software companies will take at least seven hours [33]. Meanwhile, any new virus which tries to harm the system will certainly do so without being detected.

2.5.1.2 Heuristic-based Detection

The second type of on-access scanning is heuristic-based detection, which was developed to overcome the limitations of signature-based detection. While new viruses are being discovered and analysed by the AV company, before it is able to release a signature, the user has a basic defence [5]. This type of detection monitors system behaviours and keystrokes, searching for abnormal activity, rather than searching for known signatures. Thus, some AV programs that use heuristic analysis can be used and run without updating; no action is required of either the vendor or the consumer [4].

Heuristic-based detection can thus be utilised and applied without prior knowledge of computer viruses, but it has several shortcomings, one of the most annoying of which is the creation of many more false positives than signature-based systems [1]. This is less dangerous than a false negative, but nonetheless annoying to the end-user. Such systems also need more storage space and have more effect on the system performance. Their final disadvantage is that in order to perform the heuristic analysis, extra code is needed; besides a third-party component, such as protocol parsers, needs also to be included. As a result, buggier code and increase vulnerabilities [5]. While heuristic-based detection can be applied without prior

knowledge of computer viruses, it still needs previous knowledge of the vulnerability [34].

2.5.1.3 Other Types of Detections

Among the many different techniques that have been used to solve the problem of computer virus detection, most have failed, have offered no advantages over existing ones or cannot be used in the real world due to their impracticability, such as file integrity checking [4]. Such unsuccessful techniques have not been discussed in this chapter.

2.6 Behaviour-based Virus Detection

In behaviour-based detection, a program can be identified as a virus or not by inspecting its execution behaviour [1, 35]. Unlike traditional detection techniques which rely on signatures, in behaviour-based detection, normal and abnormal measures are used in order to determine whether or not the behaviour of a running process marks it as a virus [36]. When unusual behaviour is observed, the execution of the program will be terminated. Morales et al. [23] state that despite its drawbacks, including false positives, behaviour-based detection is still the most encouraging technique, especially in dealing with novel and anonymous viruses. Therefore, behaviour-based detection has been chosen as the topic of this research.

Ellis et al. [37] used behavioural signatures in order to improve the automatic detection of worms. Signature-based detection searches for fixed regular expressions in payloads. Instead, and at a higher level of abstraction, behavioural techniques

detect patterns of executional behaviour. Ellis et al. [37] define behavioural signatures as the description of aspects of any specific behaviour of worms which are common across the manifestations of a particular worm in which its node is spanned in a temporal order. Even if a worm has not been released previously, a behavioural signature can be used to detect common implementations and the design of a worm. In general, three characteristic patterns in a network identify worm behaviour. The first is when similar data are sent between two machines, the second is when tree-like structures are observed to proliferate and the third is when a server changes into a client. Ellis et al. [37] used the notion of network application architecture (NAA), which affects the sensitivity of behavioural signatures, as an approach to distribute network applications.

It is much more challenging if an attacker wants to evade the behavioural signature, because a fundamental change in behaviour is needed, rather than only in its network footprint, which is a way of knowing the system's vulnerabilities and trying to find a method to intrude into the system. In order to detect worms, [37] placed constraints on network traffic which are violated by worm traffic patterns; these violations have proven to be straightforward to detect. They used the Abstract Communication Network (ACN) model, which is a network theoretical approach to computer networks and related data flows. The NAAs, behavioural signatures and worm propagation network are all performed within the framework of the ACN. Then, in order to identify the spreading of worms across a network, the propagation of a worm is built. The result of worm spread is the capture of a communication pattern,

which is identified by the offspring relation between nodes in the spanning trees of worm propagation. Two things were improved by this paper: first, the detection of worms can now be done without previous knowledge of worms; second, the work has shown an improvement in worm detection sensitivity.

Even if [37] can be considered as behaviour-based malware detection, its approach failed in the detection of unknown malware as stated by [38, 39] due to the fact that its approach relies on signatures to detect malware. Therefore, it can be said that the main purpose of behaviour-based virus detection is to detect anonymous malware which is missing in [37] approach. The other limitation is that large amounts of state information about network host behaviours need to be maintained by the behavioural techniques. This could be quite expensive in practice [40].

Later, Morales et al. [23] argue that detecting viruses in terms of their behaviour does not need any subsequent training analysis of known viruses and this means that less database will be needed; therefore, less storage space will be used. Their approach relies on detecting the behaviour of file viruses by their attempts to replicate. They apply runtime detection by monitoring executing processes that attempt to replicate. The behaviour of the virus is characterised by a property called self-replication, which happens when a process (virus) refers to itself (known as a transitive relation) during its attempt to replicate in read and write operation. Morales et al. use this property to distinguish between non-malignant processes and viruses.

Implementation is done by a runtime monitoring prototype called SRRAT, focusing on the tracking of Kernel mode system services and system user mode Win32

API calls. Despite the fact that the approach used in [23] has been shown to be good at detecting known and novel viruses without the need for prior knowledge of previous viruses, this detection technique may be bypassed by various viruses which replicate outside, across other directories within the same operating system. Furthermore, it can be argued that the definition of self-replication in Morales et al.'s approach is not complete due to the fact that their results have shown that there are a huge number of viruses in their analysis which did not follow their theory [41]. In addition, [23] approach lacks parallel detection in which they have two separate detections, one at the user level and the other at the kernel level, leaving the system too busy as claimed by [42]. The present research has the advantage of being able to observe both user level's API calls and kernel level's Native API calls at the same time. Moreover, the comprehensive attempt of a computer virus to attach itself to another file will be analysed in this research as discussed in Chapter 4.

2.6.1 Behaviour-based Vs. Heuristic-based detection

It has been argued that heuristic-based detection is similar to behaviour-based detection, the technique used in the present research. In fact, there is a grey area between the two detection techniques, but they differ substantially in their functionalities and ways of detecting viruses. Heuristic products check the code itself, trying to match it with known malware in order to detect new variants, whereas behaviour-based detection looks for the actions carried out by a program, intervening when it observes malevolent behaviour [35].

The present research can be used to solve the problems associated with the heuristic-based virus detection by tracking the lists of API calls that conducted in this research. By providing a precise definition of virus behaviours, the problem of false positives can be solved as will be explained in Section 7.3.1. In addition, behaviour-based detection used in this research does not require more space, as will be explained in Section 7.4. As a result, less space is needed and therefore, less effect on the system performance. Furthermore, due to the fact that behaviour-based virus detection used in this approach does not require a third party component, there is no need to the extra code. Therefore, the vulnerability problem associated with heuristic-based detection can be solved.

2.7 Related Research

2.7.1 Motivation

The study of computer viruses and their potential for infecting a computer system is active research, especially in the area of detecting anonymous viruses. Efficient implementation techniques have been submitted by many recent works to enhance their performance. Despite the fact that some of the submitted new ideas might enhance the detection of computer viruses based on their signature, at the same time their inability to detect novel and unknown viruses make them inappropriate for dealing with daily and new threats [32]. Even if the signature-based approaches try to deal rapidly with the unknown viruses by analysing them and updating their database, this solution is not perfect due to very expensive damage that can happen to the

system during the update, and hence, the system has already been inflicted by the virus [43]. Altaher et al. [44] state that “The inability of traditional signature- based malware detection approaches to catch polymorphic and new, previously unseen malwares has shifted the focus of malware detection research to find more generalised and scalable features that can identify malicious behaviour as a process instead of a single static signature” [44]. On the other hand, heuristic-based detection techniques have not provided a good solutions due to the fact that they produce many more false positives than signature-based systems. Besides, they need more storage space and have more effect on the system performance. Hence, detecting computer viruses in terms of their behaviour will help with understanding their actions, resulting in detecting unknown and newly released viruses that are a threat to computer systems every day with a better system performance.

Various frameworks have been proposed by researchers to prove that beha viour-based virus detection can deal with unknown viruses [22, 45, 46, 47], but these are still hard to understand and have some disadvantages. Moreover, some of the proposed frameworks use more than one database that is updated when a new virus is received, and thus the same problem associated with traditional antivirus software aforementioned is still unsolved. However, some of these approaches concentrate on only either the user or the kernel level of Windows operating system and this single concentration might result in infecting the system as will be explained in the next subsection.

2.7.2 API Related Work

Skormin et al. [45] designed an approach that intercepts API calls while a program is running. They detect any attempt by a malware to self-replicate at run-time. Their methodology was to trace the behaviour of normal processes and analyse API calls issued by each of them along with their input, outputs argument and the execution result. The replication of a process was modelled by the Gene of Self-Replication (GSR) based upon building blocks. Each block in the GSR is considered as a portion of the self -replication process which includes seeking files and directories, writing to files, reading from files, and closing and opening a file. This approach might detect several viruses from different classes, but on the other hand, they intercepted Native API calls in the kernel. As observed by [48, 49] Native APIs are not fully documented and that means that some viruses exist which might use some of these undocumented APIs and attack the system. In addition, Skormin [41] states that “while the number of malicious computer programs that could be written is infinite, the number of ways to implement self-replication is very limited".

Later, Alazab et al. [22] used a static analysis in order to track API calls. They analysed malware to classify executable programs as normal or malicious. They plugged in the disassembler, IDA Pro [50] in their own Python program to automatically extract API calls. They examined groups of virus steps such as search, copy, delete, read and write. They found that read and write files were mostly API calls used by malwares to infect the program. However, Zwanger and Freiling [51] stated that due to the fact that Alazab et al. [22] based their approach on IDA Pro in their

detection, they can only deal with user level's PE files [51]. Therefore, there are some viruses that might not be detected by [22] because they directly call the kernel by using Native API calls as mentioned by [49] and in their approach they only intercept user API calls.

Recently, Veeramani and Rai [46] used statistical analysis for Windows API calls to describe the behaviour of programs. They used an automated framework for analysing and categorising executables that rely on their relevant API calls. They tried to increase the detection rate by using a Document Class wise Frequency feature selection (DCFS) measure by getting the information related to malware from the extracted API calls. They categorised malware into groups and the relevant APIs were extracted from these categories. DCFS based feature selection measure is used to classify the executable as malicious or benign. In the [46] approach, they used a static technique to analyse malware however, as stated by Bayer [52], due to the nature of computer viruses, they can be designed to obfuscate the static analyser. Therefore, it can be said that there might be something missing during their analysis. In addition, their analysis and detection have been done at the user level leaving the system liable to viruses that can directly contact the kernel [49].

That means that [22, 45, 46] approaches might have a number of false positives and negatives because they rely on either kernel or user level [48, 49]. This problem can be solved by combining and tracking the Native and Win32 API calls coming from the user and kernel level that will be used in this research and explained in detail in Chapter 4.

Latterly, Ravi and Manoharan [47] proposed a system which utilised Windows API call sequence. They used a statistical model called 3rd order Markov chain to model API calls. Their system comprises 3 stages: Offline, Online and Iterative learning stages. The Offline stage subsequently comprises dataset, API call tracer, API index database, signature database, rule generator and rule database. In addition, the online stage respectively comprises the target process, API call tracer, API index database and the classifier. Finally, in the iterative learning phase, after each classification, the API call sequence and the classification label of the target process is repetitively added to the signature database to enhance the training model. It can be shown that [47] used two different databases in their approach, namely, database signature and the API index database. The API calls are represented using integer IDs and then stored in the API index database. In addition, the signature database stores both the API call integer sequence and the corresponding label of all the samples in the dataset.

They claim that their detection accuracy is better than several related approaches to their work. However, they used more than one database to store their information to catch malware. This may be acceptable as long as the detection rate is high. On the other hand, they have two main drawbacks as mentioned earlier. Firstly, they lack the detection of novel and unknown viruses which is why the behaviour-based virus detection was introduced [1, 53]. Secondly, they just intercept Windows API calls at the user level and never monitor the kernel level Native API calls, leaving their system liable to malware that directly contact the low level of Windows operating

system [48]. These shortages mean that their system has no advantages over the traditional signature-based virus detection.

2.8 Summary

Behaviour-based virus detection is a very topical subject area. It has been developed to overcome the problems associated with traditional signature-based virus detection. In this chapter a comprehensive description of computer viruses with the differences between them and other types of malicious software have been presented. The well-known signature-based virus detection was detailed with its pros and cons. In addition other techniques of virus detection with their positive and negative effects in computer systems have been provided. This chapter has concentrated more on behaviour-based virus detection as it is the main topic of this research. Different works which have used this technique to detect computer viruses have been discussed in this chapter. In addition, the system service, known as API, which will be used to analyse and trace computer viruses in this research has been discussed. Finally, related work to our research which has used this system service has been described and criticised.

The next chapter will be the preliminaries chapter. It will provide an overview of the language that will be used in our research known as Interval Temporal Logic (ITL) alongside its syntax, informal semantics with its executable subset Tempura and its semi-automatic tool AnaTempura. Moreover, related work that has used similar formal languages will be explained in the next chapter with a comparison between their and our approach.

Chapter 3

Preliminaries

Objectives:

- Provide a background about temporal logics.
- Discuss Interval Temporal Logic, its executable subset Tempura and AnaTempura.
- Provide the reasons for choosing ITL.
- Discuss and criticise related approaches.

3.1 Introduction

As previously mentioned, a computer virus will go through several steps in order to infect an operating system. The steps that represent virus behaviour need to be classified and expressed. Interval Temporal Logic (ITL) can be used to formulate all desired behavioural properties (i.e., steps of virus behaviour in our model). Therefore, ITL has been chosen to be the formal language that will be used in the present research to express the steps of virus behaviour. The existence of Tempura which is the executable subset of ITL makes it a very suitable language to be used in the present research. In addition, Tempura provides an executable framework for developing and experimenting with suitable ITL specifications. Therefore, Tempura will be used in this research to check whether a good or bad behaviour occurs with help of ITL description and system traces.

This chapter provides a background about temporal logics and then explains different temporal logics. At the same time, a comparison between these temporal logics and ITL will be presented. Subsequently, the ITL with its syntax, informal semantics, the executable subset Tempura and its syntax, and the semi-automatic tool AnaTempura will be described in detail in Section 3.3. Then, the reasons for choosing ITL to be the description language of the present research among other types of temporal logics will also be provided. Finally, related approaches which use similar tools will be explained and then criticised in Section 3.4.

3.2 Temporal Logic Background

Temporal logic is a term used to describe any system rules and symbolism. In terms of time, a temporal logic is used for representing, and reasoning about, qualified propositions [54]. Statements of time can be represented in a temporal logic, such as "She is *always* thirsty" "She will *eventually* be thirsty", or "She will be thirsty *until* she drinks something". Latterly, computer scientists and engineers have found that temporal logic is very suitable to state the expected properties of software and hardware systems [55]. For example, one may wish to state that an access is eventually given, whenever a request is made, but it is never given to two requests at the same time. These kinds of statements can appropriately be expressed by a temporal logic [56]. Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) are two successful temporal logics which have been extensively used in industrial applications [57]. Therefore, these two temporal logics will be described in the following subsections and later will be compared with our chosen one which is ITL.

3.2.1 Linear Temporal Logic

Linear-time temporal logic or linear temporal logic (LTL) is a temporal logic that models time as a sequence of states [58]. A computation path or just a path is simply representing this sequence of states. Several paths which represent different futures are considered due to the fact that the future is generally undetermined in LTL. Hence, one of these different paths will be the actual path. Formulae about the future of

paths can be encoded in LTL. For example, a condition will *eventually* be false, or *until* another event becomes false, a condition will be true.

The syntax of LTL is built upon three categories, namely, temporal model operators, a finite set of propositional variables (PA), and logical connectives or operators such as, $\wedge$ and $+$. In addition, LTL uses a fixed set atoms of atomic formulae such as (r, q, p, … or P_1, P_2,…). The atoms can represent atomic descriptions about a particular system. For instance, '*Process 1108 is suspended*, 'or *'Printer P6 is busy'*. Based on a particular interest in a system, these atomic descriptions can be chosen [58].

The figure below gives the informal semantics of the temporal operators which can only be represented by LTL as a sequence of state [59]. □ means (*always*), ◊ represents (*sometimes*), U intends (*until*), and ○ symbolises (*next*):

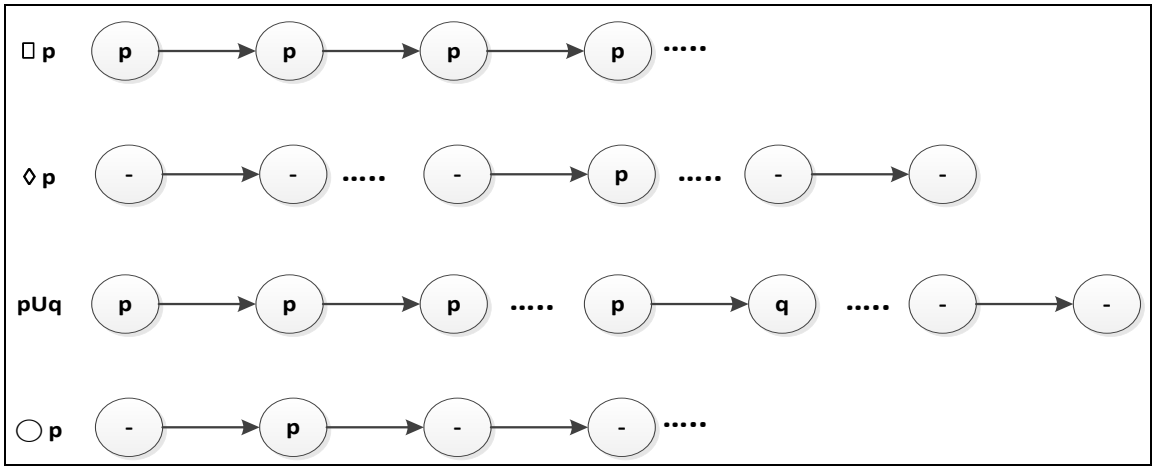


Figure 3.1: Semantics of LTL**.**

3.2.2 Computation Tree Logic

Computation tree logic's (CTL) model of time is a tree-like structure (branching-time logic). Similar to LTL, the future is also undetermined in CTL and this means that

several paths which represent different futures are built and one of these different paths will be the actual path. CTL can be used in both software and hardware tools as a formal verification. In software applications, it can be used as a model checker to check for the system safety and liveness properties.

An (**Aμ**) operator in CTL means that starting from the current state, the **μ** hold on all paths, whereas (**Eμ**) operator means that starting from the current state, exists at least one path, where the **μ** holds. However, there are similarities between LTL and CTL in that most properties expressible in LTL can also be expressed in CTL but both of them have their own unique expressible properties.

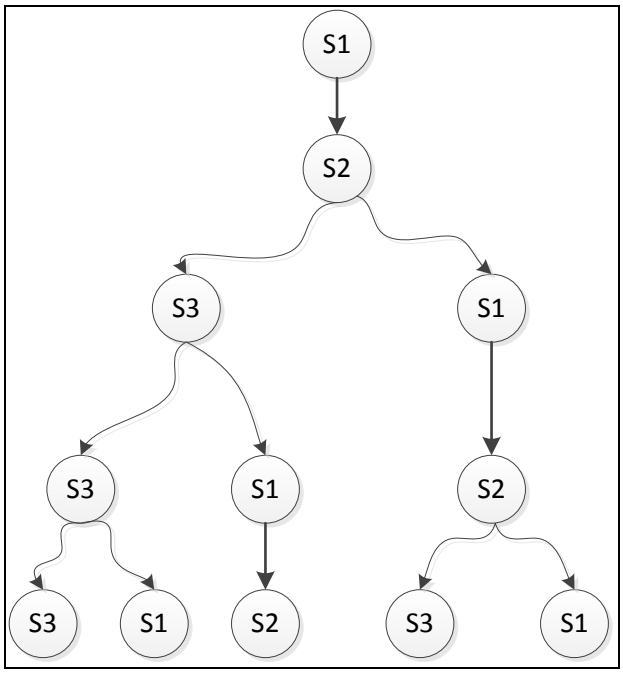


Figure 3.2: CTL Example.

3.3 Interval Temporal Logic

As noted in Chapter 1, ITL will be used in this research and our choice of this logic is inspired by the existence of Tempura, an executable subset of ITL that is a

programming language based on ITL [60]. In addition, ITL is very suitable to describe system traces, i.e., it can be used to describe bad and good behaviours.

ITL is a temporal logic whose key feature is its intervals, each of which must be a non-empty, finite sequence of states $\sigma_0\sigma_1\ldots\sigma_n$. Any ITL model has two sets, namely, a set of variables *Var* and a set of values *Val.* A state is a mapping from *Var* to *Val* denoted as *State: Var* to *Val*. ITL is known as a linear-time temporal logic for finite intervals with a discrete model of time. An interval has a finite number of states which starts from σ_0 and the length $|\sigma|$ of an interval is the number of these states minus one. For example, if an interval has states $\sigma_0\sigma_1\sigma_2$, the length of this interval is given by $|\sigma|$= (3 *states* - 1) = 2; therefore, $|\sigma|$= 2. However, a one-state interval, which is known as an empty interval, has the length zero. The sequences of states from a given system can be represented as the behaviour of this system. All the possible behaviours of a system denote the specification of this system and can be represented by ITL formulae, as explained in the next subsections [61].

3.3.1 ITL Syntax

The syntax of ITL is described in Table 3.1, in which z is an integer value, a is a static variable that does not change within an interval, A is a state variable that can change within an interval, v is a static or state variable, g is a function symbol and h is a predicate symbol.

Table 3.1: The Syntax of ITL.

Expressions
$exp ::= z \mid a \mid A \mid g\,(exp_1,\ldots\ldots,exp_n) \mid \bigcirc A \mid fin\ A$
Formulae
$f ::= h(exp_1,\ldots\ldots,exp_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v\,.\,f \mid skip \mid f_1;\ \ f_2 \mid f^*$

The constant z is a function without a parameter which has a fixed value, such as true, false, 1, 5. A static variable is one whose value remains unchanged in all states within an interval. On the other hand, a state variable is one that can change within an interval. A function symbol can be one of several operators such as, +, -, and * (multiplication), etc. An expression of the form $fin\ A$ is called a temporal expression [62]. $\bigcirc A$ means the value of A in the next state. $fin\ A$ means the value of A in the final state.

Relation symbols such as $=$ and $\leq$ are used to construct atomic formulae, which will then be composed with first order connectives such as $\neg$, $\forall$ and $\exists$ and with *skip,* *chop,* and *chop-star*, which are known as temporal modalities.

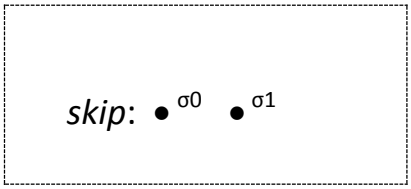
3.3.2 ITL Operators

As ITL is an extension of propositional and first order logic, it uses their connectives, such as $*$, $+$,$\neg$, $\forall$, $\exists$, $\wedge$ and $=$. It also uses the temporal operators: *skip*, ";" (*chop*) and "*" (*chop-star*) and has additional derived temporal such as: "◯" *(next*) and "□" (always).

3.3.3 Informal Semantics

Normally, expressions and formulae in ITL are evaluated over the whole interval. If there are no temporal operators in a formula, it is called a state formula. A state formula within an interval is required to hold at the initial state of that interval and it can also be expressed to hold in all intervals. For example, " □ *w* " where *w* is a state formula that holds in all states of the interval. The informal semantics of the most interesting temporal constructs are defined as follows [61]:

- *skip*: is a unit interval that has a length equal to 1.



Here is a two-state Interval that has the length of 1.

- The formula $f_1; f_2$ is known to be true over an interval if it can be decomposed (chopped) into two parts, a prefix and suffix interval, such that f_1 holds for the former and f_2 for the latter.

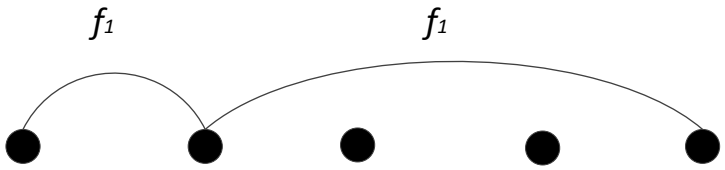


Figure 3.3: Chop Operator.

- The formula f^* which holds for an interval is true over this interval if it can be decomposed into a finite number of intervals and the subformula is true over each of these chopped intervals.

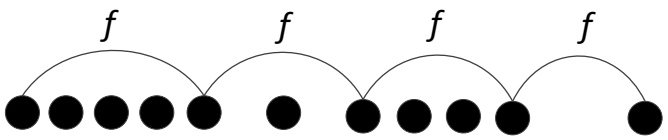


Figure 3.4: Chop-star.

3.3.4 Examples

The following are some examples of formulae used in ITL:

- The formula $S = 1$ for an interval means that the value of S in the initial state of this interval is equal to 1.
- The formula *skip;* $S = 6$ for an interval means that the value of S in the second state of this interval is equal to 6.
- The formula *skip;* $S = 2$; $S = 3$ for an interval means that the value of S in the second state is equal to 2 and the value of S is equal to 3 in any of the following states.
- The formula $\neg(true; S = 0)$ for an interval means that the value of S will never be equal to zero within this interval. Therefore, this formula is equivalent to the formula $\Box(S \neq 0)$.
- The formula $(S = 2) \wedge \bigcirc (U = 4)$ for an interval means that the value of S is equal to 2 in the first state and the value of U is equal to 4 in the second state.

3.3.5 Derived Constructs

The following constructs will be used frequently. Non-temporal derived constructs are listed in Table 3.2 and temporal derived constructs in Table 3.3, while Table 3.4 lists some program like derived constructs.

Table 3.2: Non-temporal derived constructs**.**

true	$\hat{=}$	$0 = 0$	true value
false	$\hat{=}$	$\neg true$	false value
$f_1 \vee f_2$	$\hat{=}$	$\neg (\neg f_1 \wedge \neg f_2)$	or
$f_1 \supset f_2$	$\hat{=}$	$\neg f_1 \vee f_2$	implies
$f_1 \equiv f_2$	$\hat{=}$	$(f_1 \supset f_2) \wedge (f_2 \supset f_1)$	is equivalent to
$\exists v \,.\, f$	$\hat{=}$	$\neg \forall v \,.\, \neg f$	exists

Table 3.3: Temporal derived constructs.

$\bigcirc f$	$\hat{=}$	*skip ; f*	next
more	$\hat{=}$	$\bigcirc$ *true*	non-empty interval
empty	$\hat{=}$	$\neg more$	empty interval
$\Diamond f$	$\hat{=}$	*true ; f*	sometimes
$\Box f$	$\hat{=}$	$\neg\Diamond \neg f$	always
$\textcircled{w} f$	$\hat{=}$	$\neg \bigcirc \neg f$	weak next
$\textcircled{i} f$	$\hat{=}$	*f ; true*	some initial subinterval
$\boxed{i} f$	$\hat{=}$	$\neg(\textcircled{i} \neg f)$	all initial subinterval
$\textcircled{a} f$	$\hat{=}$	*true; f ; true*	some subinterval
$\boxed{a} f$	$\hat{=}$	$\neg(\textcircled{a} \neg f)$	all subinterval

Table 3.4: Some concrete derived constructs.

fin f	≙	□*(empty ⊃ f)*	f holds in the final state
halt f	≙	□*(empty ≡ f)*	f holds exactly in the final state
keep f	≙	▣ *(skip ⊃ f)*	f holds for all unit subintervals

Therefore, our system specifications which we inferred from the observed virus behaviours will be converted to ITL formulae which then will be compared with the system behaviours using AnaTempura (discussed in the next subsections) in order to identify any malicious behaviour observed in the system.

3.3.6 Our Choice of ITL

There are a number of reasons that support our choice of ITL over other temporal logics. These reasons make ITL a suitable language to express the behaviour of a virus.

- An Interval represents a behaviour (trace) of a system. A set of intervals (behaviours) denotes all possible behaviours of that given system. Therefore, an interval will describe a virus behaviour and the set of these behaviours will be compared with the normal system behaviours to ensure that the system is safe.
- Once ITL formulae are constructed, its executable subset Tempura makes them ready to be executed, i.e., ITL formulae can be refined into Tempura code and then can be programmed and executed. This leads to a rapid prototyping and debugging of the desired system. As a result, the advantage of the refinement of ITL formulae into Tempura code, make it possible to check for virus actions (behaviour) practically at runtime.

- The order of virus actions (behaviour) is significant in this research. In fact, it will play a crucial role in distinguishing between benign and viral processes. Moreover, ITL offers syntactic constructs like chop to describe the order of API calls as described in the next chapter. Therefore, this advantage gives us the ability to ensure that the order of computer viruses' actions is met and detected.

These reasons make ITL a powerful language for describing viruses because distinguishing between normal and malicious behaviour can be highly accurate due to intervals representation. The following subsection will discuss our executable language Tempura.

3.3.7 Tempura

Tempura is a language which is the executable subset of ITL; that is, once a formula is given, Tempura generates a satisfying interval for that formula. In addition, an ITL formula can be executed by the interpreter of Tempura if it satisfies the three following conditions [60]: the formula is deterministic, the length of the interval is known and the values of the variables are known in all parts of this interval. Tempura has both state and static variables which can have primitive types such as *booleans* and *integers*, and derived types like lists. In addition, Tempura shares most of the features of other imperative programming languages; for example, it has lists that are similar to arrays and vectors in other languages. In addition, it has the regular operations over expressions, such as *, +, mod, =, and, or. It also has the ability to provide for the rapid development, testing and analysis of Tempura specifications.

Indeed, ITL has an advantage over other temporal logics when used with its subset Tempura, because it provides fast and convenient testing during execution [63]. The Tempura tool will be used to check whether a good or bad behaviour occurs with help of ITL description and system traces.

3.3.8 Syntax of Tempura

As aforementioned Tempura shares most of the features of other imperative programming languages and hence it can be considered as a programming language that based on temporal logic. In addition, each state in Tempura needs to be defined and taken into account. Moszkowski [60] reports that the syntax of Tempura is divided into three categories, namely, locations, expressions and statements [60]:

3.3.8.1 Locations

Values in Tempura are stored and examined in a *location*. For example, in Listing 3.1, the variables *A* and *B* are permissible locations.

Listing 3.1: Tempura code.

```
/* run */ define test() =
{ exists A, B :
  {
  A=4 and B=1 and
  halt(A=0) and (A gets A-1) and (B gets 2*B) and
  always output(A) and always output(B)
  } }.
```

The above code includes three statements, *halt* which means that the code will stop running when *A* equals to 0. Secondly, *gets* in the first formula means that *A* will be subtracted in each of the following states by 1, and in the second formula means that *B* will be multiplied by 2 in each of the next states. Finally, *always* means that the output of both *A* and *B* will be displayed in each state as shown in the execution below in Figure 3.5.

```
Tempura 42%
State   0: A=4
State   0: B=1
State   1: A=3
State   1: B=2
State   2: A=2
State   2: B=4
State   3: A=1
State   3: B=8
State   4: A=0
State   4: B=16

Done!  Computation length:  4.  Total Passes:  5.
Total reductions:  124  (124 successful).  Maximum reduction depth:  7.
Tempura 43%
```

Figure 3.5: Execution of Listing 3.1.

3.3.8.2 Expressions

In Tempura, expressions can be either an arithmetic expression or a boolean expression. For example, +, −, ., ÷, and *mod* can be considered arithmetic operations, when they are placed between arithmetic expressions such as ($e_1 \div e_2$ or e_1 *mod* e_2). On the other hand, boolean expressions can be constants such as *true* and *false*, and the temporal constructs such as *more* and *empty* can also be considered as boolean expressions.

3.3.8.3 Statements

In Tempura, statements can be either *simple* or *compound*. Simple statements can be built from the construct shown below [60].

true	(any-operation)
false	(abort)
$l = e$	(simple assignment)
empty	(terminate)
more	(do not terminate)

On the other side, compound statements can be expressed like parallel composition (^), implication (⊃), weak next (ⓦ), always (□).

3.3.8.4 The Operator Chop and Loops

Loops in Tempura are similar to those in other related programming languages. For example, loops such as, *for, repeat and while* can be defined in Tempura in different ways as shown in Listing 3.2. In addition, Tempura has the loop '*chopstar* { ... }' that is equal to '*while true do* { ... }' as shown in the listing below.

Listing 3.2: Loops in Tempura.

```
while <condition> do { ... }

repeat { ... } until <expression>

for <variable> < <integer expression> do { ... }

for <variable> in <list/string expression> do { ... }
```

```
for <integer expression> times do { ... }

chopstar { ... }
```

The chop ";" operator in Tempura, stand for sequential composition. Intuitively, the term ";" means "followed by". The following Tempura codes and execution will illustrate how this operator can be used in Tempura.

Listing 3.3: Chop in Tempura.

```
/* run */ define test1() = {

exists I: {

{

 {len(3) and I=0 and I gets I+1} and always output(I) ;

 {len(4) and I gets I+1} and always output(I)

}}}
```

Listing 3.3 shows how the ";" operator can be used in Tempura. The above Tempura code is equivalent to the one in Listing 3.4. In addition, they will provide the same execution output as shown in Figure 3.6. However, with the chop operator, the formula before the semicolon must define an interval length. For example, if the length in the formula before the semicolon in Listing 3.3 was not defined, it will not be executable, since it could satisfied by any number of behaviours.

```
Tempura 84%
State   0: I=0
State   1: I=1
State   2: I=2
State   3: I=3
State   3: I=3
State   4: I=4
State   5: I=5
State   6: I=6
State   7: I=7

Done!  Computation length:  7.  Total Passes:  8.
Total reductions:  116  (116 successful).  Maximum reduction depth:  8.
```

Figure 3.6: Execution of Listing 3.2.

For more information on Tempura and examples, we refer the reader to [60, 61]. However, the tool that will be used in our system (AnaTempura) will be explained in the following subsection.

Listing 3.4: Equivalent to Chop in Tempura.

```
/* run */ define test2() = {

exists I: {

{

{len(7) and I=0 and I gets I+1} and always output(I)

}}}
```

3.3.9 AnaTempura

AnaTempura is a semi-automatic tool that is used to perform runtime verification of a system. AnaTempura has the following features:

- It offers specification support; and

- It provides validation and verification support in the form of simulation and runtime testing in conjunction with formal specification.

AnaTempura is a development of C-Tempura which was introduced by Roger Hale and is now maintained by Antonio Cau and Ben Moszkwoski [61]. The runtime verification technique uses assertion points to check whether a system satisfies timing, safety or security properties expressed in ITL. The assertion points are inserted in the source code of the system and will generate a sequence of information (system states), like values of variables and timestamps of value change, while the system is running. [61].

AnaTempura has an open architecture that is known to be pluggable tool, i.e., it allows new tool components to be plugged in. The general system architecture of AnaTempura that is used in our system is shown in Figure 3.8. Inputs to the system are the system behaviours and Tempura specification with the desired properties. The output will be a result that indicates whether the desired properties are satisfied by the system behaviours. The role of AnaTempura in our research is to examine the satisfaction of the system behaviours with the virus properties. If the system behaviour satisfies a virus property, then an infection has occurred and needs to be detected, otherwise, the system is uninfected.

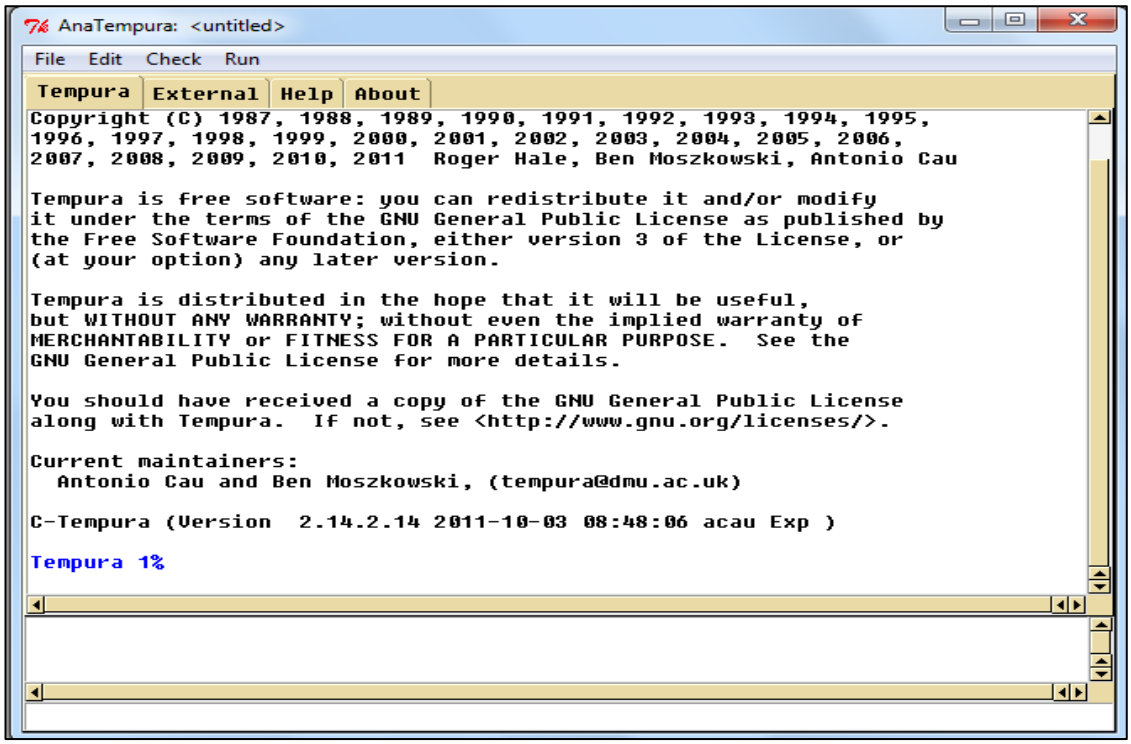


Figure 3.7: AnaTempura Interface.

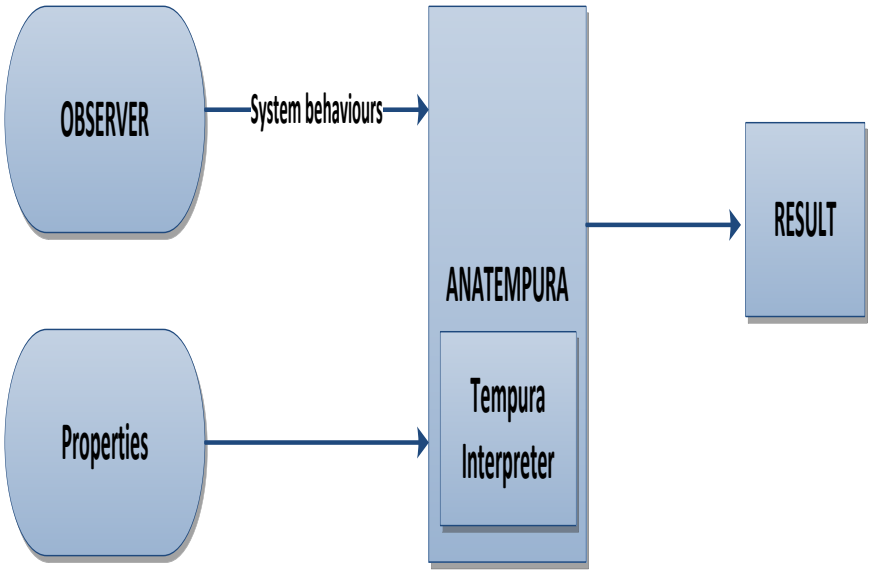


Figure 3.8: The general system architecture of AnaTempura.

A behaviour in this approach represents a sequence of states (i.e., an interval). A property is a set of behaviours. In addition, a system is also a set of behaviours that should satisfy a property (That is to say, for all behaviours σ of the system, $\sigma \in$ Prop, where Prop represents a property). Figure 3.9 shows an overview of the system

analysis in AnaTempura. The first step is to formulate all desired behavioural properties (i.e., virus behaviours in our model) which are stored in Tempura files. The second is to observe the system behaviours. Finally, the process (AnaTempura) decides whether or not the behaviour satisfies the desired properties.

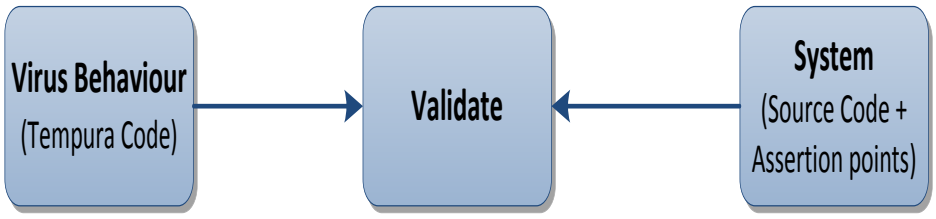


Figure 3.9: The analysis process.

3.4 ITL Related Work

Ye [64] reports an anomaly detection technique to detect intrusions in network systems. He used a Markov Chain model in order to create a temporal profile of normal behaviours in a network or computer system. Ye claims that the temporal behaviour profile, which is defined as the temporal profile of an action sequence, is significant in detecting intrusions, because intrusions consist of sequences of connected computer activities. He defines the Markov Chain model as a non-continuous random process indicating how variables change randomly at different points in time. The normal profile can be identified by learning the normal behaviour of the system from historic data. In this model, the probability of intrusion is inferred by analysing the observed behaviours of the system, supported by the Markov Chain model. If a probability has low support, this suggests an anomalous behaviour which may result in intrusion.

The technique used in [64] was tested and implemented on the Solaris operating system, which has a security extension called the Basic Security Module (BSM) that records events by monitoring activities in a host. A number of event types were gathered by BSM and the normal and intrusion activities were inferred from these events. Ye built a long-term normal profile of temporal behaviour, then to detect a significant difference, the temporal behaviour in the recent past was compared with the long-term normal profile. This technique was found to have high sensitivity and specificity, in other words, low rates of both false negatives and false positives. In addition, temporal behaviour data can be taken from a large-scale domain such as a network or from a small one such as a file, user or special program. However, Chandola et al. [65] argue that there is a limitation with Markov Chain techniques in which they potentially require a huge amount of space in order to store all transition frequencies.

Another approach is to identify binaries of worm and viruses by using a combination of reverse engineering and model checking [66]. This approach comprises three stages, beginning with the classification of worm and virus behaviour, by identifying five functions which describe their activity and then detecting malicious properties. The second step is the model checking process, described by [57] as an automated technique that produces a finite-state model of a system and a logical property, then systematically checks whether this property holds for a given initial state in that model. The final stage is to use linear temporal logic (LTL) in order to encode the malicious behaviour of worms and viruses.

In the approach of [66], a given worm program behaviour is characterised by a set of executions, then LTL formulae are used to encode this set of executions. Formal

specifications are used and the elements of computer virus programs are identified in order to characterise malicious behaviour in worms. The binary program is translated into a finite model representation and then given to a model checker for verification purpose. The authors claim that even if there are different virus source codes, they still have the same operational behaviour; thus the method used is beneficial, since it will succeed in capturing malicious behaviour. It can be argued that the approach of Singh and Lakhotia [66] will succeed as long as all worms and viruses have the same operational behaviours. On the other hand, the work of [1, 5] shows that there are complex viruses which can change their functional and operational behaviours. Therefore, there are several viruses which can evade the Singh and Lakhotia method of detection as described in [67]. The approach adopted here, using ITL, has the advantage of being able to define more complex temporal features [63] and simultaneously has an intuitive advantage over approaches based on LTL that has been used in attack detection. This intuitive advantage of ITL indicates that an interval clearly describes the sequential nature of system behaviours.

Later, Holzer et al. [68] have used a verification technology in order to identify and discover malicious software. Their approach relies on formalising malicious behaviour using Computation Tree Predicate Logic (CTPL) that has been extended from the classic CTL. The CTPL is used because it has predicates that enable the malicious behaviour to be concisely formalised. A model checker called *Mocca* is used and this model checker expects an input to be a plain text assembly source code. At first, they make sure that the program is unpacked in which static analysis is used to do the unpacking. Then, a plain binary is the outcome of unpacking the program that will be

disassembled in order to construct the assembly source code which is used as an input to *Mocca*. Next, the assembly file will be parsed by *Mocca* in order to generate an abstract model of the executable.

A structure called Kripke structure is derived in order to syntactically model the assembly code. Every instruction in the code is represented by a predicate and its parameters are considers as constants. In the code, each line is corresponded to a state that is only identified to a specific location. Finally, the model checker *Mocca* reports whether or not the assembly code satisfies their system specification. That is, whether the file is a malware or not. The paper of [68] has produced an optimised research in the use of formalism for the purpose of detecting computer malwares but unfortunately, there are too many translation steps which are not needed before checking whether it is a malware or not. It also uses the binary code prior to checking has been used. In addition, it also lacks the mechanism that can perform the runtime verification of the system. The runtime verification drawback of [68] work can be solved by our integrated workbench, AnaTempura.

Naldurg et al. [69] propose a framework for intrusion detection using temporal logic. Their approach relies on runtime monitoring of temporal logic specifications, using a logic called EAGLE with three temporal operators: next time, previous time and concatenation. This logic supports finite trace monitoring and allows the pattern of security attacks to be formally specified. In order to determine whether the specification is violated or not in this technique, [69] used an online monitoring algorithm to automatically match the absence of an attack specification with traces of system execution, sounding an alarm whenever the specification is violated. The idea

behind this proposal is to produce temporal formulae which involve statistical predicates for the expected behaviour of security-critical programs, monitoring the system execution in order to check whether the formula is violated. If the formula is violated by the observed execution, then an intrusion has happened. Therefore, attacks can be detected even if they are previously unseen. A prototype called MONID was used, allowing the system to be implemented in an online or offline fashion and to detect intrusion.

Munoz et al. [70] report that there is a drawback in [69] approach in which the distinction between an intrusion and normal behaviour is not clear [70]. In addition, ITL, which will be used in this research, differs from and has advantages over the logic used in [69] and other logics in that it has the sequential chop operator ";" that composes two phases together which can also be used to remember the order of virus actions. In addition, ITL has a tool called Tempura which allows our system specifications to be validated, prior to the real implementation, which is done through a simulation and animation. It also provides a fast prototyping and debugging for our system ITL specifications [60]. In addition, ITL is very suitable to describe system traces, i.e., it can be used to describe bad and good behaviours. The Tempura tool will be used to check whether a good or bad behaviour occurs with help of ITL description and system traces.

3.5 Summary

In order to infect a system, a virus will carry out different actions. These actions need to be traced and then expressed. This chapter presented the language which will be

used to express these virus actions (steps). First, a background about different types of temporal logics was discussed. Interval Temporal Logic was explained in detail alongside with its syntax, informal semantics, the executable subset Tempura and its syntax, and the semi-automatic tool AnaTempura. After that, the reasons which support our choice of ITL amongst other temporal logics were justified. Furthermore, related researches which utilised similar languages or tools to express virus behaviour were explained and compared with the present research.

AnaTempura cannot be used on its own to detect computer viruses. In other words, an extension to AnaTempura is needed to handle virus detection. Therefore, there is a need for other tools to be integrated with AnaTempura. These tools should be able to deliver the sequences of API calls to AnaTempura. These tools will be explained in Chapter 5.

The next chapter explains the methodology of this research. Two architectures will be used in this research namely, virus behaviour analysis and virus detection. Virus analysis includes tracking API calls of normal and malicious programs beside the steps that a virus carries out in order to attach it to another file. On the other hand, the virus detection architecture will be used to detect these viruses and to distinguish them from benign processes.

Chapter 4

Framework for Behavioural Detection of Viruses

Objectives:

- Provide a general overview of the proposed framework.
- Describe the virus behaviour analysis, including the tools which have been used to trace API calls.
- Provide the virus detection architecture.
- Describe the observed virus behaviour in ITL.

4.1 Introduction

This chapter explains what has to be done in order to fulfil the requirements of this research. It begins with a brief account of the methods used to gather and analyse data on computer viruses, then sets out the framework used to achieve the desired detection. Next, it explains every component of the framework, how it was used in the present investigation and why it was chosen. The chapter ends with a list of API calls representing virus behaviour and how they can be detected using Interval Temporal Logic (ITL).

4.2 Main Framework

As this research investigates behaviour-based virus detection, selected types of virus were analysed in order to discover how they behave, i.e., what they do inside the system. Their behaviours were observed as a sequence of steps in order to track them. To infect a system, a virus will go through several steps, which together constitute its particular behaviour. From these steps we can infer specifications which constitute the virus behaviour description that matches our initial system requirements, as shown in Figure 4.1. From this specification, ITL formulae (to be discussed later) are derived. These formulae can be inferred from the virus behaviours that have been analysed.

Once the ITL formulae have been derived, one of the inputs of our system will have been constructed. The other requirement of our system is that these formulae must match system behaviours. To ensure this we created a monitoring algorithm using AnaTempura (discussed later) to match input2 (system behaviours) with input1

(ITL formulae) in order to see whether input2 satisfies input1. If input2 satisfies input 1, then an infection has occurred. This is the agreed main framework that was used in order to fulfil the requirements of this research.

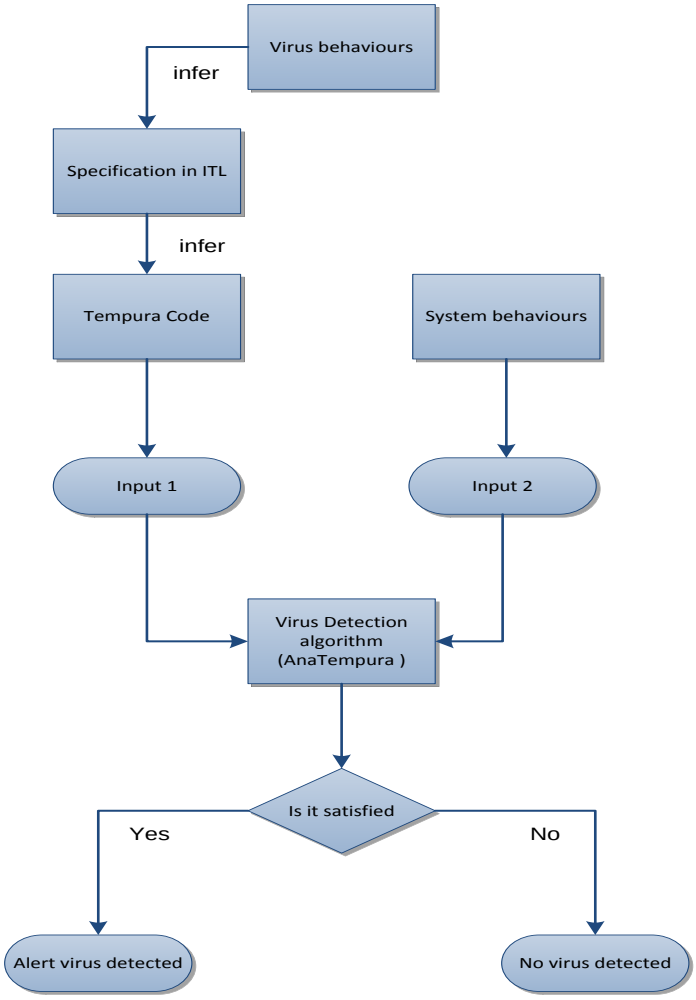


Figure 4.1: The main framework.

4.3 Virus behaviour

It is very important to understand application program interfaces (APIs) and their features as well as the portable executable (PE) file format, in order to trace the behaviour of programs and to understand hidden features of malicious codes. Therefore, an outline of PE and API calls is provided here in order to enhance

understanding of these important system services. An application programming interface (API) is a system service that has been built to help software components to communicate with each other and it acts as an interface between these components. APIs may operate for object classes, variables, data structure, and routines. Their specifications can take many forms, including an International Standard such as POSIX. They can also be vendor documentation such as the Microsoft Windows API, or the libraries of a programming language, e.g., Standard Template Library in C++ and Java. However, a PE file is divided into sections and each section supplies different information about the PE file, like, for example, the number of API calls imported, number of dlls, file headers etc [22].

4.3.1 Win32 Portable Executable Format

Microsoft introduced the Win32 PE format as the standard executable format for all its Windows operating systems in all supported processors [35]. As the present research is concerned with extracting and understanding API calls, it is essential to have an understanding of the PE format.

PE has its own structure, as shown in Table 4.1, which gives a good understanding of what a PE file looks like, involving the DOS headers and the PE headers. The signature of the PE file and file properties such as timestamp and the number of sections are the two types of data that the PE header starts with. The section table has four sections, beginning with the .text section, which is known as the code section. The .data section involves the original entry point (OEP), which points to

the beginning of the execution of a PE file. It also includes the writable global variables [2]. Finally, the .rdata section contains read-only data [35].

Table 4.1: Portable Executable Structure [22].

Section Table
DOS Header
COFF File Header
Optional Header
Standard Fields
NT additional Fields
Optional Header Data Directories
Export Table
Import Table
Resource Table
Exception Table
Certificate Table
Base Relocation Table
Debug
Architecture
Global Ptr
TLS Table
Load Config Table
Bound Import
Import Address Table (IAT)
Delay Import Descriptor
COM+ Runtime Header
Reserved
Section Table
.text
.rdata
.data
.idata

The most important element of the PE structure for this research is the IAT section, which includes the addresses of all the functions imported by a file, including API calls, which are used in this research in order to determine how viruses behave. A more detailed explanation of how the IAT is used in this research will be given in the next chapters.

4.3.2 Windows Application Program Interface Calls

In 1995, Microsoft released Windows 95 and at the same time introduced a set of system calls known as Win32 API, which represented a 32-bit application program interface [60, 69]. The new APIs had the advantage of higher system speeds because they provided a set of optimised system operations [57]. User applications in the Windows operating system (OS) based on these API function calls are stored in dynamic link libraries (dlls) such as User32.dll, Kernel32.dll, Advapi32.dll, and Gui32.dll, in order to gain access to system resources involving registry and network information, processes and files [35]. According to [71, 72], the functionality provided by the Windows API can be grouped into eight categories:

Base Services

Access to the fundamental resources available to a Windows system can be provided by the base services. The fundamental resources may be processes and threads, devices, file systems and error handling. These base services reside in kernel.exe, krnl286.exe or krnl386.exe files on 16-bit Windows, and in this approach target, known as 32-bit Windows, they reside in kernel32.dll.

Advanced Services

Access to functionality additional to the kernel is provided by advanced services. The additional functions to the kernel may be the shutdown/restart the system (or abort), Windows registry, manage user accounts and start/stop/create a Windows service. These advanced functions on 32-bit Windows reside in advapi32.dll.

User Interface

The functionality to create and manage screen windows and the most basic controls, such as, receive mouse and keyboard input, buttons and scrollbars and other functionalities associated with the GUI part of Windows, are provided by the user interface APIs. On 16-bit Windows, these functions reside in user.exe, and on 32-bit Windows, they reside in user32.dll. The basic controls and the common controls (Common Control Library) reside in comctl32.dll on Windows XP versions.

Graphics Device Interface

The functionality for outputting graphical content to monitors, printers and other output devices is provided by the Graphic Device Interface. On 16-bit Windows, they reside in gdi.exe and on 32-bit Windows in user-mode, they reside in gdi32.dll. GDI for Kernel-mode is provided by win32k.sys which communicates directly with the graphics driver at kernel level.

Common Dialog Box Library

Applications for the standard dialog boxes for saving and opening files, choosing font and colour etc; is provided by Common Dialog Box Library. On 16-bit Windows, the library resides in a file called commdlg.dll, and on 32-bit Windows it resides in comdlg32.dll. It is grouped under the User Interface category of the API.

Windows Shell

Applications are allowed by the component of the Windows API to access the functionality provided by the operating system shell, as well as to enhance and alter it. On 16-bit Windows, the component resides in shell.dll, and on 32-bit Windows, it resides in shell32.dll. In both 16-bit and 32-bit, shlwapi.dll has the Shell Lightweight

Utility Functions. These functions are grouped under the User Interface category of the API.

Network Services

Access to the different networking capabilities of the operating system is given by network services. Its sub-components include Winsock, NetBIOS, RPC, NetDDE and many other sub-components.

Common Control Library

Access to some advanced controls provided by the operating system is given to applications by the Common Control Library. These may be progress bars, toolbars status bars and tabs. On 16-bit Windows, the library resides in a DLL file called commctrl.dll, and on 32-bit Windows, it resides in comctl32.dll. The library is grouped under the User Interface category of the API.

Each Win32 API call has its own memory address place in the import address table (IAT) which every process in the system has and which each process will consult when it makes an API call, as shown in Table 4.1. A Win32 API call is normally called from a process running at the user level [22], then the called API will be handled by the system and converted to its equivalent function, known as a Native API call, which will be understood by the kernel of the OS. A service in the kernel will handle the requested operation and its outcome will return to the original user application that made the call [35].

The majority of systems services run at the kernel and need privileges in order to access it. Native API calls, which can be directly called by any process at the kernel

level, are dealt with in the dynamic link library (ntdll.dll) in order to have the kernel provide the requester service [35].

The complete list of kernel mode functions is stored with memory location addresses in the system service dispatch table (SSDT), which is accessed each time a Native API routine is called. The parameters are then passed to the memory location and the function continues with its execution [3, 9].

The following will give an example in order to have a better understanding what is the idea of system calls in the operating system and how it can be represented in the code. This C code has issued a *CreateFile* system call which will be redirected to its appropriate system kernel library in order to create the file as shown in Listing 4.1. It can be seen in the code this system call will create a text file called “myfile” which has the following characteristics. It is a general read file that can be shared as a read file with attributes set to normal as indicated by the code.

Listing 4.1: CreateFile C code.

```
hFile = CreateFile(
        TEXT ("myfile.txt"),            // Open myfile.txt
        GENERIC_READ,                   // Open for reading
        FILE_SHARE_READ,
        NULL, // No security
        OPEN_EXISTING,                  // Existing file only
        FILE_ATTRIBUTE_NORMAL,
        NULL);                          // No template file
```

4.3.3 Static and Dynamic Analysis

There are two popular techniques to analyse computer viruses, namely, static and dynamic analysis. According to Bayer [52], static analysis is the process of analysing an executable code without practically executing it. To analyse a code using static analysis several steps need to be carried out. Firstly, a binary is generally disassembled, i.e., converting the binary code into its matching assembler instructions. Afterwards, conclusions about the functionality of the code can be drawn by the aid of both the control flow and data flow of the instructions. Many static binary analysis techniques [73, 74, 75] have been established to help the detection of various types of malware. There are a number of pros and cons for static analysis. One of the biggest advantages that static analysis provides is that it is able to examine the complete code even faster than its corresponding dynamic one. One of the shortcomings of static analysis is that many interesting questions can be asked about the code and its properties which unfortunately remain unanswered. Due to the fact that it is not easy to analyse malicious software, it can be designed to use binary obfuscation techniques that are used to prevent the static analysis approach from accurately analysing and dissembling program codes [52].

On the other hand, dynamic analysis techniques are the opposite of static analysis in which they analyse the code during the execution of the program. In the dynamic techniques, only the executed instructions by the code are analysed. Hence, dynamic analysis solves the obfuscation problem which thwarts static analysis techniques. The environment that is used to dynamically analyse the sample is always

questionable. Undoubtedly, running a malware in a physical computer that might be connected to the Internet might harm other machines as the malware has the ability to spread. In addition, running an isolated machine which needs to be re-installed every time a dynamic test is carried out would be insufficient due to the overhead that is involved.

Using a virtual machine, i.e., a virtualised computer such as Oracle VM VirtualBox [76] is a popular choice. Therefore, the tested malware cannot influence the physical PC only the virtual one. The virtual machine allows the user to have a clean state of the operating system, known as a *snapshot,* the user has the ability to restore to this clean state each time a malware test is completed. This solution is known to be faster than using an isolated machine because it is easier to restore a virtual machine than install an operating system in a real PC. One main disadvantage of virtualisation is that the malware analysed might determine that it is executed on a virtual machine and harm the real one. There are a number of works [77, 78] that have demonstrated how a program can determine whether it is running in a virtual machine or not. This problem can be solved by installing different operating systems in the host machine as explained in Section 4.2.1.

To achieve a better understanding of computer viruses, and due to the fact that our approach used behaviour-based virus detection, dynamic analysis has been considered to be the main analysis technique used in the present research. There has also been used static analysis to gain a first impression about what types of API calls a virus might issue when infecting a system and attaching itself to another file.

4.3.4 Virus Analysis

This section explains the methodology used to extract API calls which are used to represent the behaviour of a virus. Figure 4.2 shows the mechanism used to analyse and extract API calls. In addition, these tools will be discussed in further detail in the following sections. Existing software was used to obtain information about the viruses through the following steps:

Step one: Unpack the virus.

Step two: Get the assembly code by disassembling the virus.

Step three: Extract the sequence of API calls that represent the virus behaviour.

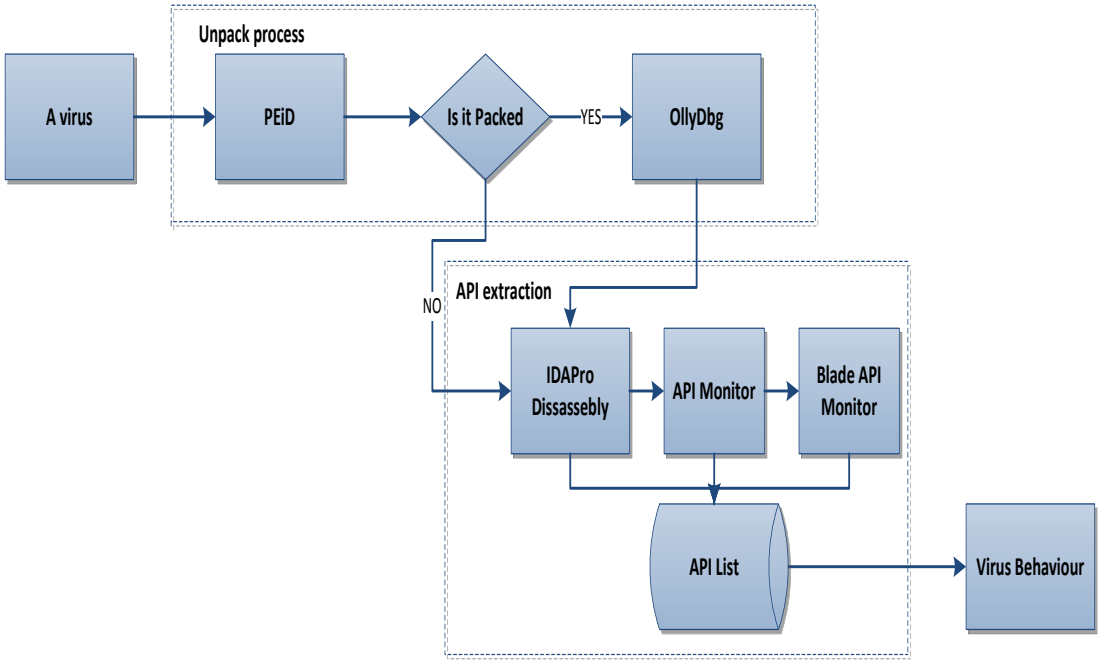


Figure 4.2: API extraction mechanism.

4.3.4.1 Build a Secure Environment

In order to analyse computer viruses, a secure environment is needed to make sure that no virus can escape the system and infect other machines. In addition, some

viruses will use the Internet or a local area network (LAN) to spread their malicious effects, allowing them to spread very widely indeed. Therefore, a virtual machine (VM) (Oracle VM VirtualBox) [76] was used in this research in order to secure the system. The Linux Ubuntu operating system was used as host with Windows XP as a guest to ensure that no viruses leaked from the guest to the host, because as explained earlier, a virus that infects one OS will not run when a different OS platform is used. In some cases, viruses will use the Internet to connect to anonymous remote hosts. It is preferable not to connect to these unknown hosts, even if the virus is running on a virtual machine, so a way to prevent this is needed. However, the behaviour of viruses is the target and the Internet plays an important role in tracking these behaviours. Therefore, a fake Internet was used, allowing all the network activities in addition to allowing the tracking of virus behaviour in this research. This was achieved without causing any risk to the real Internet by installing NetKit [79], which provides a simulation of the entire Internet. NetKit was therefore installed on the host (Ubuntu) machine and then the virtual machine ran Windows XP using the fake internet.

4.3.4.2 Unpacking the Virus

Packers are known as “anti-anti-virus” programs and also can be called “anti-reversing”, because they exist to fight against anti-virus software as well as reverse engineering techniques. Packers are mostly used to disguise and/or compress codes. According to Alazab [22], packers are just computer programs which have the ability to restore the original executable image of a file from its encrypted and compressed one

in a secondary memory location. Hence, the code might appear to do one thing, but it actually does something else, which is likely to confuse researchers.

Nowadays, computer virus writers have the benefit of using these packers to make their viruses run faster, as well as avoiding detection systems [22]. Furthermore, the methods of packing make recognising and understanding viruses very complicated both for detection systems and analysts, because the authors can make small code modifications in order to change a signature and so avoid detection. Packing also makes analysis by researchers less easy, because to extract and understand unpacked code requires a third party tool, beside a deep and strong understanding of assembly language and the kernel, which leads to a better understanding of low level programming [22].

However, a number of researchers have reported the construction of tools that automatically unpack viruses such as Eureka [80], Ether [81] and Renovo [82]. The present research uses PEiD [83] to unpack the virus samples examined. PEiD is an unpacking tool that detects most common packers, cryptors and compilers for PE files. The first step was to use an interactive disassembler, IDA Pro [50], to decide whether a virus was packed or not, after which PEiD was used to indicate which packer (e.g. UPX, Upack, Xpack or PEPack) had been used. As Figure 4.2 shows, OllyDbg was used to seek the entry point of the virus and to dump the unnecessary code. It would also save the newly unpacked virus in order to conduct a clear investigation of the malware. Figure 4.3 shows that approximately 70% of the viruses analysed in this research had been packed and needed to be unpacked, using the process explained above, while the remaining 30% were directly observed.

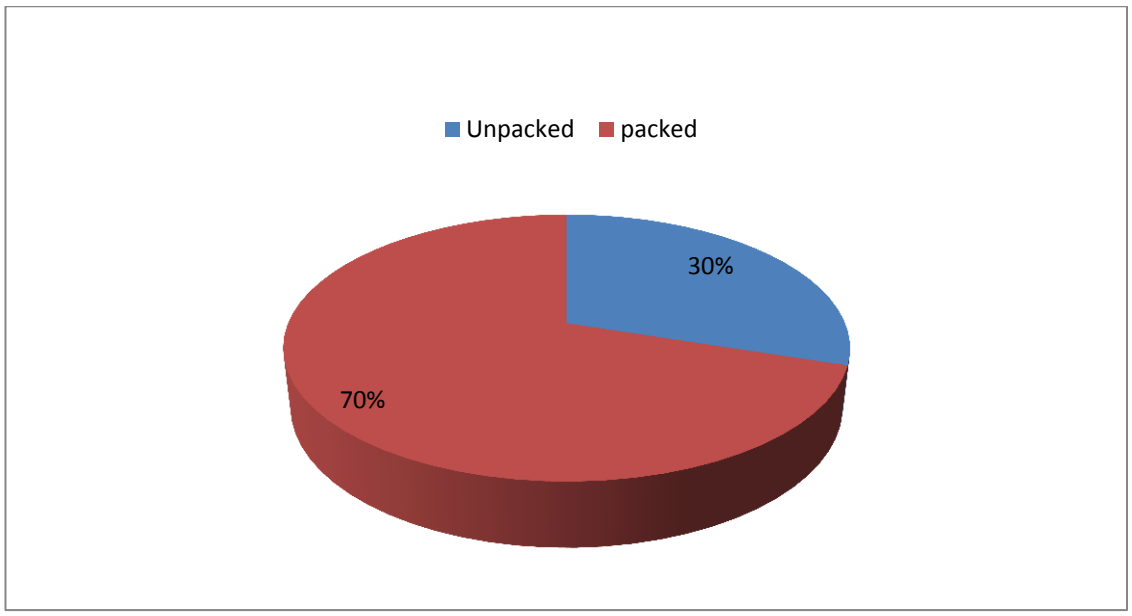


Figure 4.3: Percentages of packed and unpacked viruses analysed.

4.3.4.2.1 PEID

PEiD is the tool that is used in this research to unpack the virus samples examined. The role of this tool is to indicate whether a process is packed or not and which packer was used to pack it. As shown in Figure 4.4, a selected virus was tested to see whether it is packed or not. The reason why PEiD is used in this research is, because a clean sample is needed to carry out the analysis and especially the static analysis.

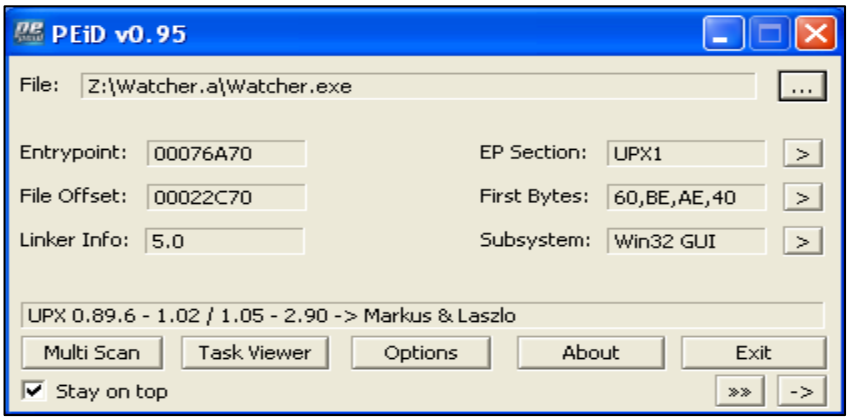


Figure 4.4: A snapshot of PEiD.

The snapshot above indicates that the “Watcher.exe” virus has been packed and the packer is “UPX”. To unpack this virus another tool (OllyDbg) is needed. Moreover, OllyDbg will be discussed in the following point to show how it can help with unpacking a selected process.

4.3.4.2.2 OllyDbg

OllyDbg is a tool that is utilised in the API extraction mechanism to search for the entry point (OEP) of the virus and to dump the unnecessary code. It would also save the newly unpacked virus in order to conduct a clear investigation of the malware. Therefore, a clean and unpacked sample can be saved and then run. As shown below in Figure 4.5, a selected virus was assembled by OllyDbg.

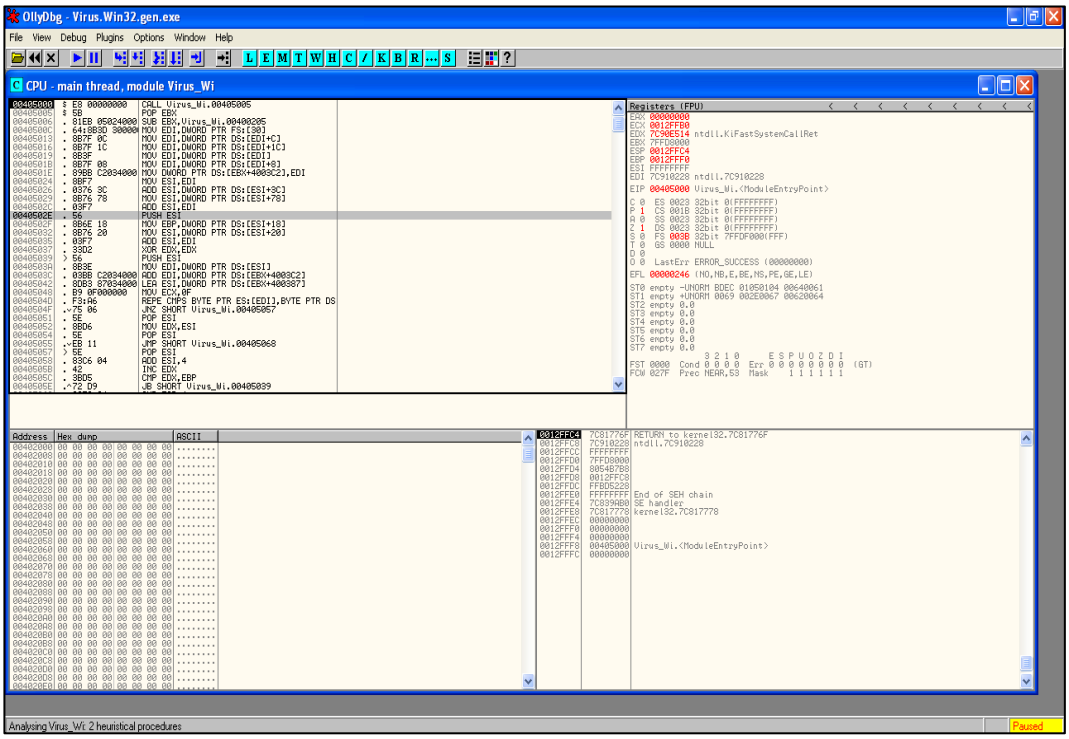


Figure 4.5: A snapshot of OllyDbg.

The above snapshot shows that the assembly code of the Windows 32 “gen.exe” was extracted by OllyDbg. Then, the unnecessary code will be removed and a clean executable virus will be saved. Afterwards, the unpacked virus can be run and tested as a normal PE file. The API calls of the clean virus will be statically analysed by IDA Pro Disassembler as discussed in the following point.

4.3.4.3 Extracting the Assembly Code (Static API Extractio)

IDA Pro can be used to extract the assembly code from both executable (such as PE, ELF, EXE, etc.) and non-executable files and is the most practical disassembly tool [22]. It runs a static analysis [50] and can detect whether a file is packed, as well as disassembling the code, thus providing more details and improving the understanding of the code.

IDA Pro was selected as a part of the API extraction mechanism used in this research because it can statically and automatically extract API calls from a file, giving an initial image of what sort of API calls the file might make. Thus, using IDA Pro allows API calls to be statically extracted and gathered, offering an important method of identifying virus behaviour. In order to have more evidence about the API calls made by viruses, IDA Pro needs to be used with more than one tool that provides tracking of API calls at runtime. Both Blade [84] and API Monitor [85] were used to extract API calls during run time (dynamically), while the virus was being executed, to give us a fuller image of the API calls made by the virus.

4.3.4.3.1 IDA Pro Disassembler

In this research, the IDA Pro Disassembler tool is used to extract the assembly code from the executable processes. As shown in the snapshot below, IDA Pro Disassembler helps us to carry out static analysis in which it allows us to gain a first impression about what types of API calls a virus might issue when infecting a system and attaching itself to another file. Figure 4.6 shows the assembly code of the normal process "iexplore.exe".

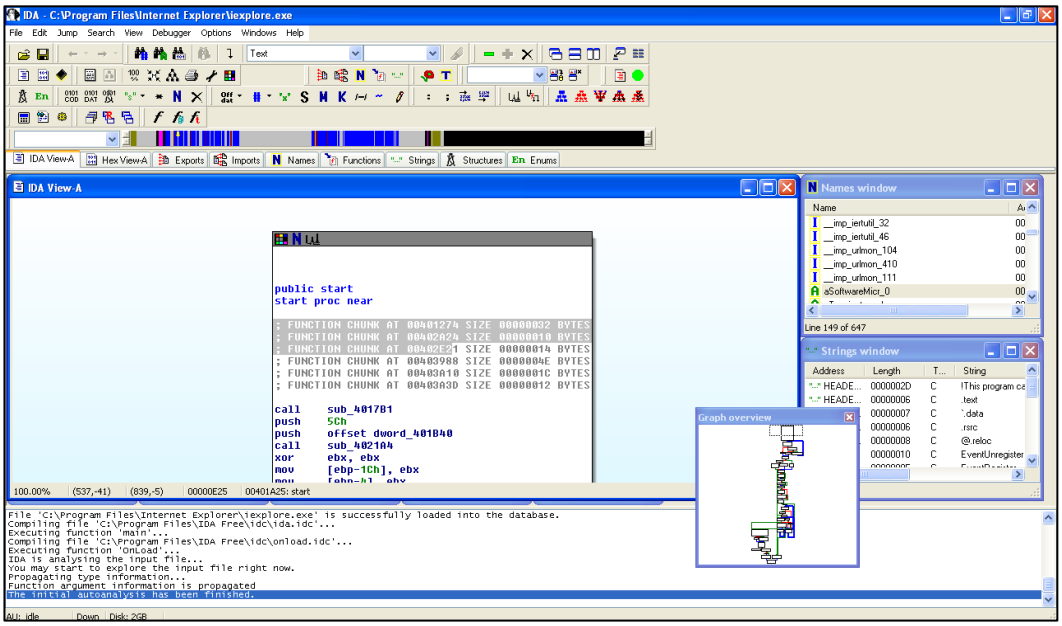


Figure 4.6: A snapshot of IDA Pro.

From the above assembly code a static analysis of API calls can be obtained. Figure 4.7 shows the list of API calls of "iexplore.exe" normal process can be statically extracted by IDA Pro Disassembler from the process's assembly code (Function calls graph).

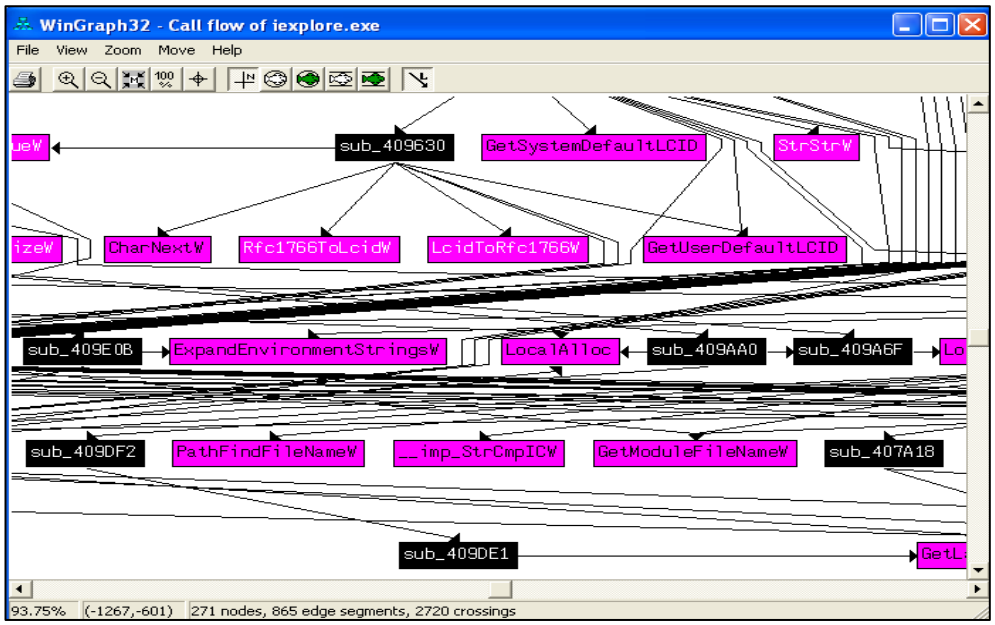


Figure 4.7: List of API calls of iexplore.exe.

4.3.4.4 Extracting API calls (Dynamic)

As explained by [23], Windows API calls play an important role in exploiting the power of Windows, allowing virus writers to use API calls to gain more security privileges and perform malicious actions. Windows APIs issue calls to perform several actions, such as user interfaces, system services and network connections, which can be utilised for good or evil [22]. Because API calls will give a full and complete description of a particular program, the analysis of its API calls will lead directly to the understanding of its behaviour.

Viruses are just like normal programs and can be distinguished by tracking their API calls that lead to malicious actions. Therefore, this research concentrates on tracing API calls in order to understand virus behaviour. As shown in Figure 4.2, more than one tool [50, 84, 85] was utilised to trace API calls in static and runtime environments. Most researchers [22, 23] rely on just one tool, which runs either

statically or dynamically, but this research uses both in order to have a full understanding of what API calls have been made and when. Static analysis misses some API calls when comparing to dynamic analysis. In addition, there are some Win32 and Native API that appear in [85] but not in [84] and vice versa. Thus, these three tools have been used in this research to track API calls.

4.3.4.4.1 API Monitor

In order to observe API and Native API calls at runtime while the virus was being executed, API Monitor is used in this research. The reason why dynamic analysis is used to extract Win32 and Native API call is because runtime analysis can help us to identify the different order of API calls that a virus normally carries out in order to attach itself to another file. Figure 4.8 shows a list of API calls that was observed at runtime by API Monitor.

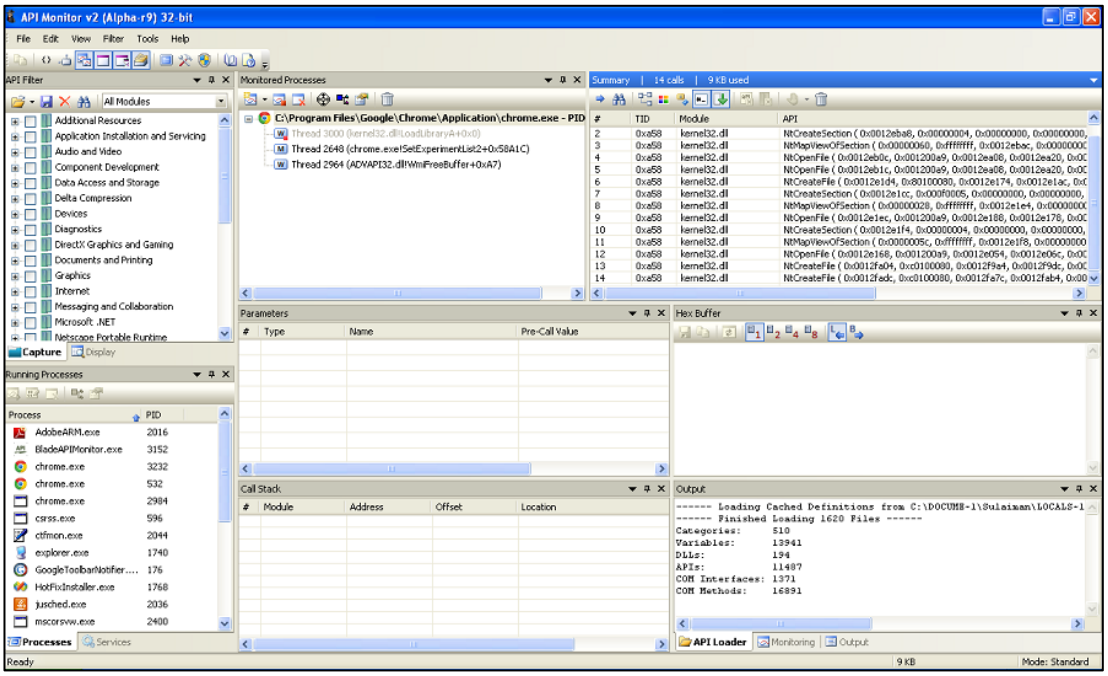


Figure 4.8: API and Native API calls at runtime.

A selected normal process “chrome.exe” was tested at runtime by API Monitor. It can be seen from the figure above that a list of API calls with their parameters can be observed while the process is running. In addition, this list can be saved anywhere in the operating system for future analysis.

4.3.4.4.2 Blade API Monitor

The Blade API Monitor is another dynamic API monitor that is used in this research because there are some Win32 and Native API calls that appear in API Monitor but not in Blade API Monitor and vice versa. Therefore, these two dynamic analysis tools have been used in this research to track API calls. Figure 4.9 and 4.10, shows how the Blade API Monitor can be used to observe API calls at runtime. Firstly, the DLL files which their functions will be monitored, need to be chosen. In our research, “kernel32.dll” and “ntdll.dll” need to be selected in order to observe both Win32 and Native API calls as shown below.

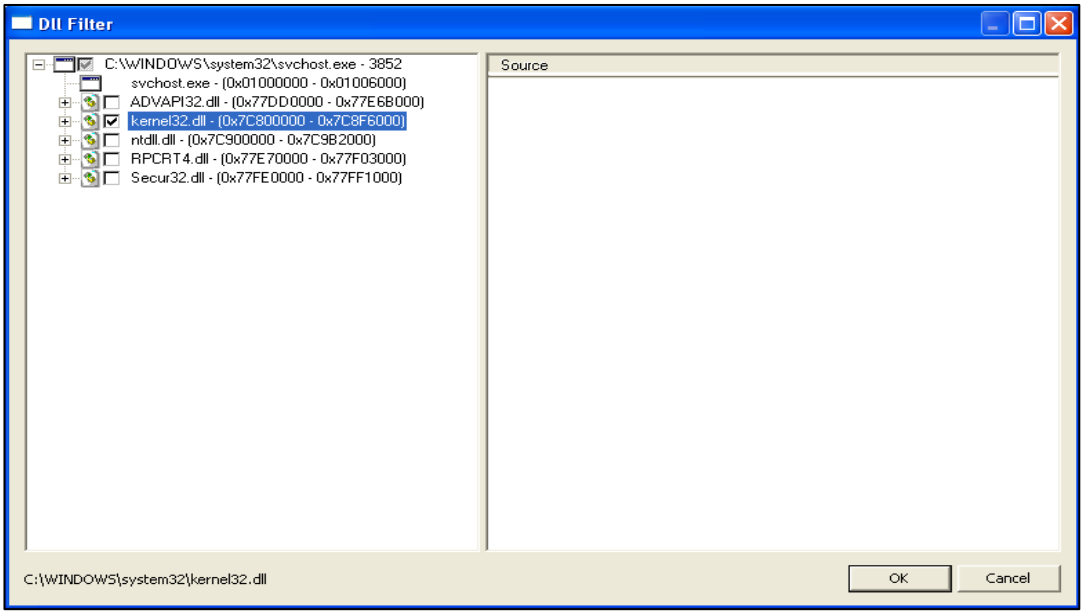


Figure 4.9: Choosing which DLL file to monitor.

The following figure shows how Win32 and Native API calls can be monitored at runtime by the Blade API Monitor.

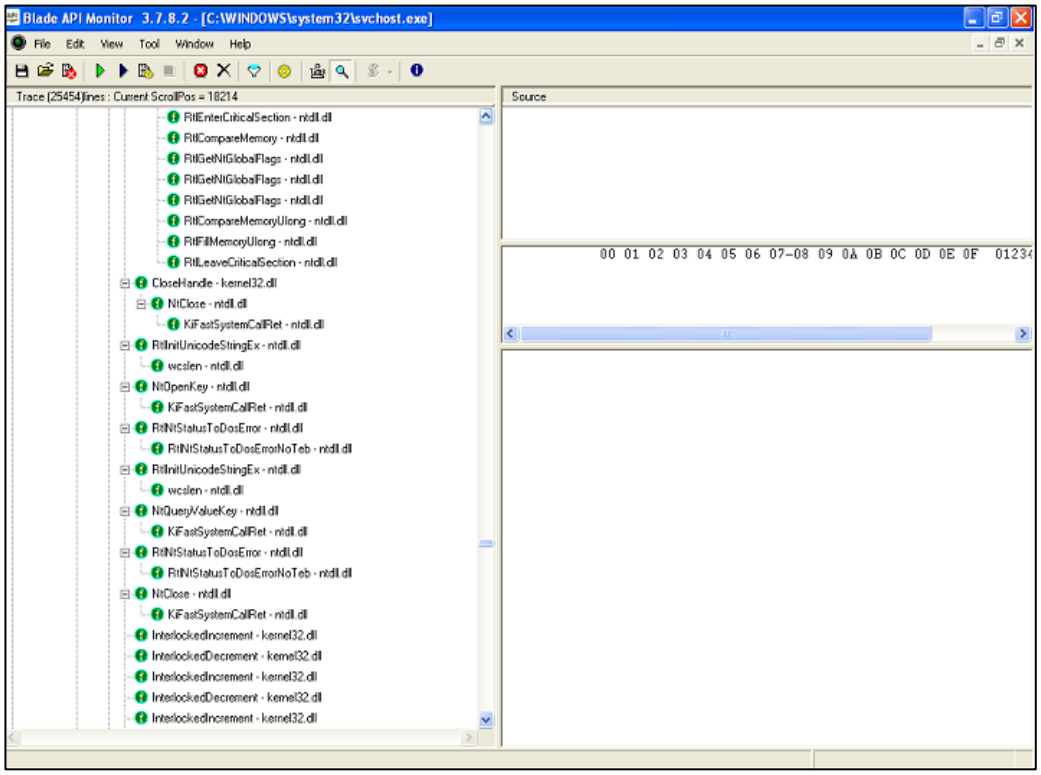


Figure 4.10: Blade API Monitor.

The present research considers API calls to provide a way to determine whether malicious actions have been performed or not, by analysing them to understand their behaviour and to indicate whether a file contains a malicious or benign program. To do this, 283 virus samples downloaded from [86, 87] and 50 Windows (XP) normal processes, such as *svchost.exe* and *iexplore.exe*, were examined to discover what sort of API calls malicious programs use in order to perform their actions.

Table 4.2: Normal Processes.

systeminfo.exe	systray.exe
alg.exe	clipsrv.exe
services.exe	msdtc.exe

imapi.exe	lsass.exe
cisvc.exe	svchost.exe
dmadmin.exe	mqsvc.exe
mqtgsvc.exe	dllhost.exe
netdde.exe	igfxsrvc.exe
firefox.exe	smlogsvc.exe
googledrivesync.exe	sessmgr.exe
locator.exe	regsvc.exe
scardsvr.exe	snmp.exe
atsvc.exe	mstask.exe
tapisrv.exe	explorer.exe
taskmgr.exe	hkcmd.exe
msiexec.exe	wmiapsrv.exe
taskman.exe	wuauclt.exe
ctfmon.exe	spoolsv.exe
winlogon.exe	csrss.exe
smss.exe	iexplore.exe
winmsd.exe	msseces.exe
chrome.exe	tracert.exe
tasklist.exe	sort.exe
rsvp.exe	ping.exe
cipher.exe	compact.exe

The research began with the assumption that a virus must read from and write to a file, as [22, 23, 45] explain, in order to infect a computer, to replicate itself, to infect other files and to spread throughout the world. More precisely, the following five steps are considered to represent virus steps in a behaviour:

1) Find to infect
2) Get information
3) Read and/or copy
4) Write and/or delete
5) Set information.

- **Find to Infect**

In order to infect, a virus needs to find a file or to retrieve the contents of a directory in which to write its malicious code. This research has concentrated on three types of

computer virus, listed in Table 4.2: those which overwrite existing files, known as overwriting viruses, those which can be attached to existing files, known as parasitic viruses, and those which create a file resembling a known one, known as companion viruses.

After analysing the API calls issued by a group of computer viruses related to the three types explained above, it has been considered that 'find to infect' as the first step in the behaviour, addressing its potential API function calls that relate a search to a particular file or directory. Table 4.3 groups the first category and its API call functions into a single list. These API calls were compared to see whether a virus had searched to infect a particular file or directory.

Table 4.3: Virus descriptions.

Virus type	Description	Behaviour
Overwriting	A virus (V) will replace its content with an existing file (F) by overwriting it.	1. Read "V.exe" 2. Open "F.exe" 3. Write "V.exe" into "F.exe" 4. Close ""V.exe""
Parasitic	A virus (V) will attach itself to an existing file (F) by injecting its code into F and replace its entry points.	1. Open "V.exe" 2. Read "V.exe" code 3. Open "F.exe" 4. Inject code into "F.exe" 5. Replace "F.exe" entry point
Companion	A virus (V) will change the name of an existing file (F) with its original name.	1. Read "F.exe" 2. Rename "F.exe" as "F.ex" 3. Rename "V.exe" as "F.exe"

Table 4.4: Find to infect API function Calls.

Behaviour type	API function calls
Find to infect	"FindFirstStream","FindFirstFileTransacted","FindFirstStreamTransacted", "FindClose","FindNextFile","FindFirstFileName","FindNextFileName", "FindFirstFileNameTransacted","FindNextStream","FindFirstFileEx", "FindFirstFile".

- **Get Information**

The second category of steps in a virus behaviour observed in this research was to discover a file's attributes, to retrieve specific information regarding a file, or to retrieve information on a directory, such as path name. A virus needs to have information about a particular file or directory to infect it and to read and write to it. Table 4.5 shows the API function calls a program can issue to get information about a particular file or directory.

Table 4.5: Get information API function calls.

Behaviour type	API function calls
Get information	"GetFileAttributesEx","GetFileAttributesTransacted","GetFileAttributes", "GetFileInformationByHandle","GetFileBandwidthReservation","GetShortPathName", "GetCompressedFileSizeTransacted","GetFileInformationByHandleEx", "GetCompressedFileSize","GetBinaryType","GetFileSizeEx","GetFileSize","GetFileType", "GetTempFileName","GetFinalPathNameByHandle","GetLongPathNameTransacted", "GetFullPathNameTransacted","GetFullPathName","GetLongPathName".

- **Read and/or Copy**

Read and write calls are the most important API calls issued by viruses, because they give it the ability to attach itself to other files and spread. As explained by [45], there is a very narrow difference between normal and malicious behaviour in the case of system calls. Indeed, although this research has given careful consideration to distinguishing between normal and abnormal activity, there exist some legitimate processes that may look like malicious software but would never be captured by the detector used here, because they will never act exactly the same as the malware, i.e., there is always a difference, however slight. Previous researches such as [23] and [45] have observed that normal processes will never issue system calls that have the same

order as computer viruses. This means that our concept of virus behaviour has to trace system calls from the beginning to the end, having a set of system calls which have to be made in a particular order, because normal processes are supposed never to follow the concept of replication completely.

Therefore, read and write function calls must be made in a certain order, i.e., a virus will read a file first and then write to this or another file. In addition, other observed API calls of read and write categories may or may not be called, but when it comes to read and write API calls, they must be called by the file for it to be considered a virus. Table 4.6 shows the API function calls a program can issue to read from or copy a file. Copy API calls are considered to be malicious here, because some viruses will copy to or from files, or create new files when they infect a system [22]. The use of 'and/or' in the category name means that a copy API call may or may not be issued by a virus.

Table 4.6: Read and/or copy API function calls.

Behaviour type	**API function calls**
Read and/or copy	["**ReadFile**", "**ReadFileEx**","**OpenFile**","**OpenFileById**","**ReOpenFile**"], ["CreateHardLinkTransacted","CreateHardLink","CreateSymbolicLinkTransacted", "CreateSymbolicLink","CreateFile","CopyFileEx","CopyFile","CopyFileTransacted"].

- **Write and/or Delete**

As mentioned in the previous subsection, a file must issue write API calls in order to be classified as a virus. Therefore, every read API call should be followed by a write API call, issued at any time by the same file, to be considered a virus and not to conflict with benign processes. Table 4.7 shows the list of API calls which a file will issue to

write to or delete a file. However, as with 'copy', the delete API call is considered malicious, because some viruses will delete some files when they infect a system, as reported by [22]. It will also be optional, as the phrase and/or appears in the category name; that is, the API delete call may or may not be issued by a virus.

Table 4.7: Write and/or delete API function calls.

Behaviour type	**API function calls**
Write and/or delete	["**WriteFile**", "**WriteFileEx**", "**ReplaceFile**"], ["DeleteFileW", "DeleteFileTransactedW","CloseHandle"].

- **Set Information**

The last category of steps in a virus behaviour observed in the research is the setting of specific information regarding a file, which leads to a change in its attributes. It has been observed that after infecting a file, a virus will need to change some of the file information in order to deal with it in the future. Therefore, this category has been considered and Table 4.8 lists the API calls that a file needs to set and change the file information.

Table 4.8: Set information API function calls.

Behaviour type	**API function calls**
Set information	"SetFileInformationByHandle", "SetFileValidData", "SetFileBandwidthReservation", "SetFileShortName", "SetFileAttributesTransacted", "SetFileApisToOEM", "SetFileAttributes", "SetFileApisToANSI","NtSetInformationFile".

Therefore, five categories of steps in representative virus behaviours, reiterated in Table 4.9, were observed in this research and were compared with API calls to determine whether a virus was present. At least two of the eight API calls

presented in bold in the table and representing the third and fourth categories, namely, *ReadFile, ReadFileEx, OpenFile, OpenFileByld, ReopenFile, WriteFile, WriteFileEx and ReplaceFile* must be called in order to say that a file is a virus. That is, at least one of the five bolded API calls in the third category must be called by a file and one of the three bolded API calls in the fourth category must subsequently be called, for that file to be treated as a virus.

Table 4.9: API function calls for categories steps.

Behaviour type	**API Function Calls**
Find to infect	"FindFirstStream","FindFirstFileTransacted","FindFirstStreamTransacted", "FindClose","FindNextFile","FindFirstFileName","FindNextFileName", "FindFirstFileNameTransacted","FindNextStream","FindFirstFileEx","FindFirstFile".
Get information	"GetFileAttributesEx","GetFileAttributesTransacted","GetFileAttributes", "GetFileInformationByHandle","GetFileBandwidthReservation","GetShortPathName", "GetCompressedFileSizeTransacted","GetFileInformationByHandleEx","GetCompressedFileSize","GetBinaryType","GetFileSizeEx","GetFileSize","GetFileType","GetTempFileName","GetFinalPathNameByHandle","GetLongPathNameTransacted","GetFullPathNameTransacted","GetFullPathName","GetLongPathName".
Read and/or copy	["**ReadFile**", "**ReadFileEx**","**OpenFile**","**OpenFileByld**","**ReOpenFile**"], ["CreateHardLinkTransacted","CreateHardLink","CreateSymbolicLinkTransacted","CreateSymbolicLink","CreateFile","CopyFileEx","CopyFile","CopyFileTransacted"].
Write and/or delete	["**WriteFile**", "**WriteFileEx**", "**ReplaceFile**"], ["DeleteFileW", "DeleteFileTransactedW","CloseHandle"].
Set information	"SetFileInformationByHandle", "SetFileValidData", "SetFileBandwidthReservation", "SetFileShortName", "SetFileAttributesTransacted", "SetFileApisToOEM", "SetFileAttributes", "SetFileApisToANSI","NtSetInformationFile".

The previous five categories were found to have used API calls that could be called by a file at the user level, known as Win32 APIs. However, there is an alternative, whereby Native API calls perform this function in order to provide the service requested by the kernel. Win32 APIs are converted to Native API calls by the ntdll.dll process [22, 45], in order to be understood by the kernel. For example,

consider that a user requests a list of files which belong to a particular directory. When a request is received form the user input, a Win32 API call is needed in order to complete this request. One of the most famous Win32 API to list files is *FindFirstFile()* and this API call is exported by kernel32.dll [88]. Firstly, *FindFirstFile()* will pass the directory name, and if the request succeed, a handle will be passed to *FindNextFile()* in order to list the remaining files in the directory [89]. Each time these functions are called, the operating system need to go through these steps, as shown in Figure 4.11. After that, *FindFirstFile()* will call *NtQueryDirectory()* Native API which is located in Ntdll.dll. The *NtQueryDirectory()* will be then sent by Ntdll.dll to the kernel-mode and executed like all other Native API calls. However, there exist some files that can call the kernel directly, avoiding the need for user level API calls [48]. These calls were observed in this research.

Table 4.10 shows the Native API calls that can be issued by a file in order to be classified as a virus. However, these Native API calls are not fully documented, as Microsoft does not make them publicly available [49], so API call researchers are struggling to acquire more knowledge about them. Therefore, both Native and Win32 API calls need to be observed and taken into account in order for the present research to achieve good results. Figure 4.12 shows the order of the five categories alongside with their description.

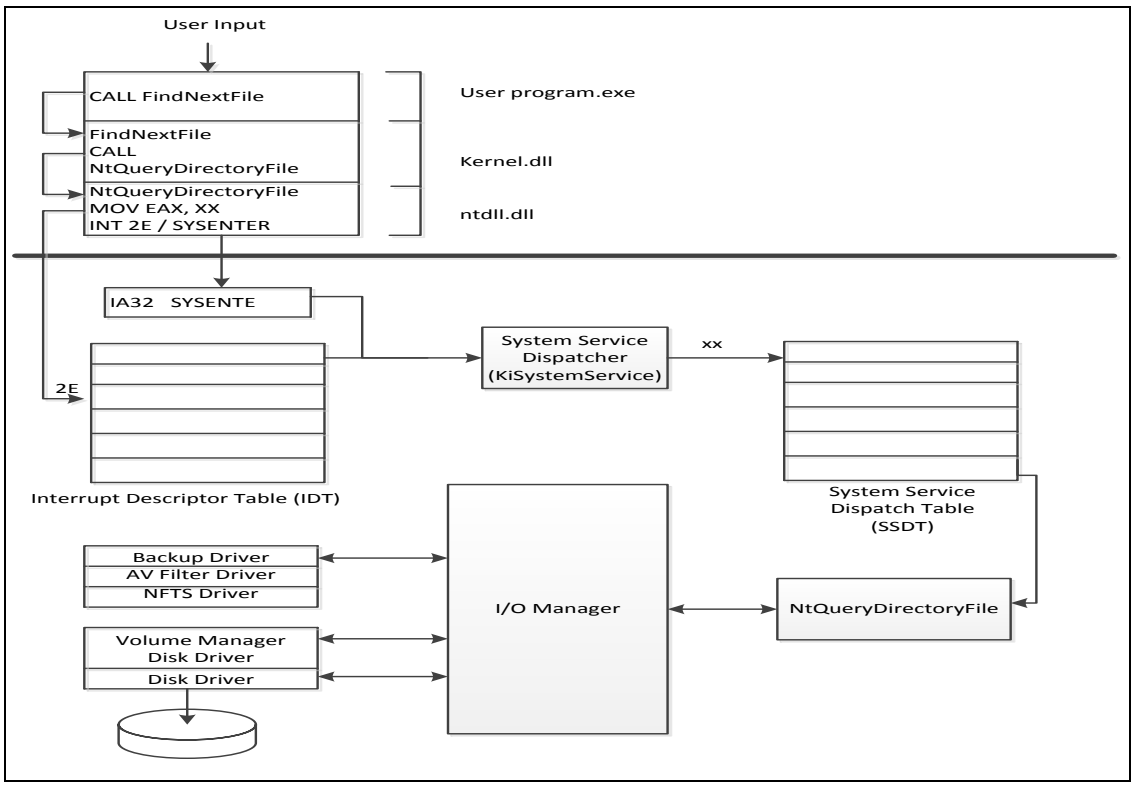


Figure 4.11: Various steps to call FindNextFile() function [89].

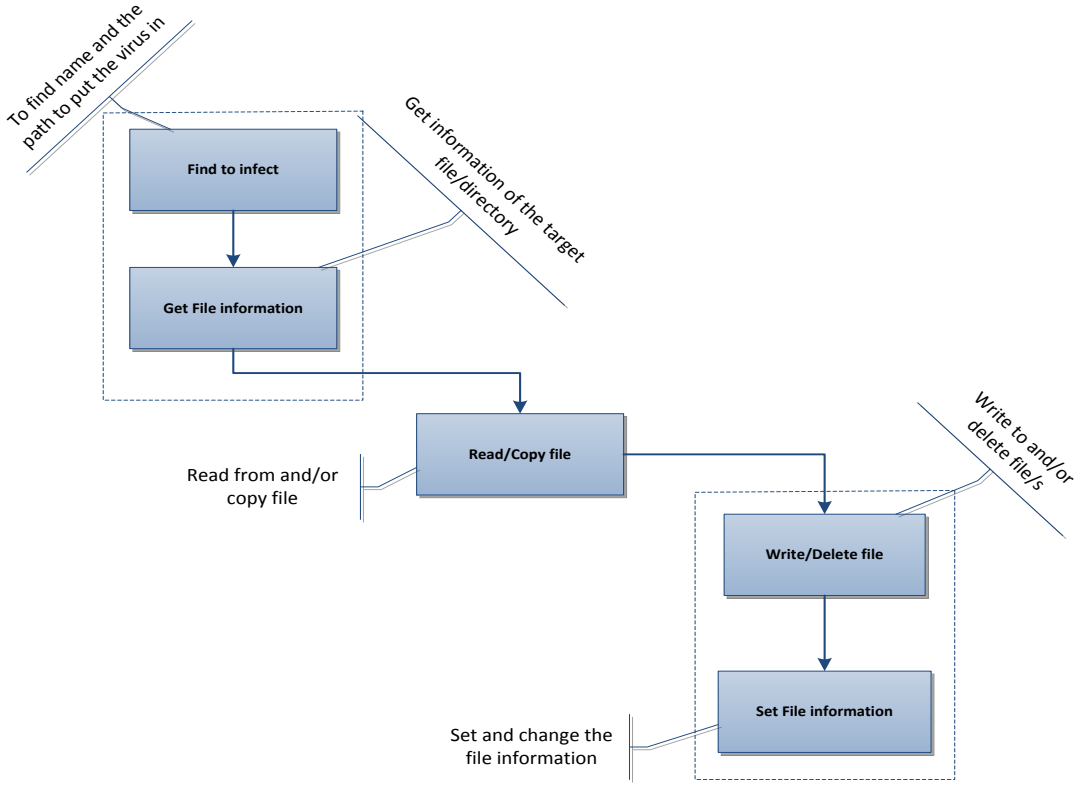


Figure 4.12: The Five Categories.

Table 4.10: Native API function calls for categories of steps.

Behaviour type	API function calls
Find to infect	"NtQueryDirectoryFile".
Get information	"NtQueryAttributesFile", "NtQueryInformationFile".
Read and/or copy	["**NtOpenFile**"," **NtReadFile**"], "NtCreateFile"].
Write and/or delete	["**NtWriteFile**"], ["NtDeleteFile", "NtClose"].
Set information	"NtSetInformationFile".

4.3.5 Virus and Normal Process API Examples:

As abovementioned, 283 virus samples and 50 normal processes that are running in Windows (XP) and Windows 7, were examined in this analysis in order to discover what sort of API calls normal and viral processes carry out and how to distinguish them. For example, when examining *svchot.exe*, we found that it never issues a read and write API calls which refer to itself.

4.4 Virus Detection Architecture

The observation of steps in a virus behaviour used in our system is based on the API hooking method at runtime. Hooking APIs provides the ability to intercept a set of API calls and redirect them to other functions [23, 48]. The benefit of doing so is to examine these calls in order to decide whether a virus is present or not. API hooking is done in either user or kernel mode.

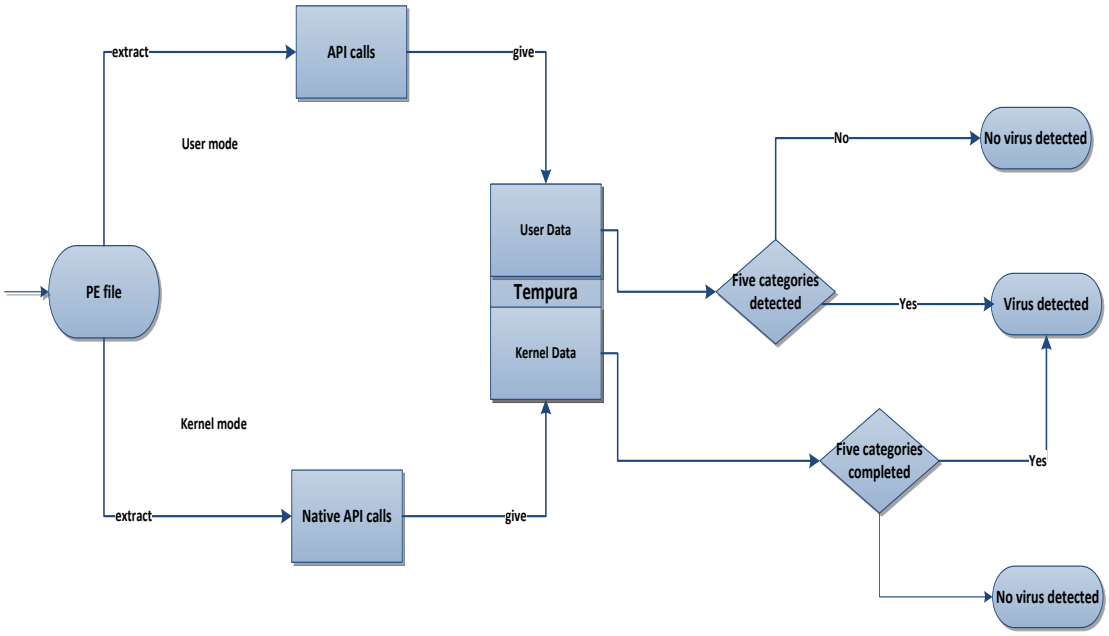


Figure 4.13: Detection Architecture.

4.4.1 User Mode API Hooking

User mode API hooking, based on the technique of altering the IAT, redirects API calls to another place [29]. All API calls are hooked at runtime by an existing tool representing our second input (virus behaviours) in Figure 4.1. However, Tempura will receive the API calls in order to decide whether a virus is present, as shown in Figure 4.13. If a virus is detected, the system will not allow it to continue making API calls and the file is terminated.

When a prototype was run in user mode only, the virus detection rate was low, because viruses are designed to evade the detection used at the user level [29]. Therefore, if no virus was detected, it was directed to the second approach and the kernel Native API calls were examined.

4.4.2 Kernel Mode API Hooking

The majority of computer viruses try to run at the kernel level in order to gain more security levels and control of the system, which cannot be gained at the user level [29]. At the kernel level, Native API hooking does not differ from the user level, at which the SSDT can be overwritten and redefined. Therefore, Native API calls will be received at runtime by Tempura, where they are examined for a virus.

If the user level fails to detect any suspicious API calls issued by a file, it is directed to the kernel level for further examination. If the Native API calls indicate that it is suspicious (i.e., a virus), it will be terminated, while if no suspicious behaviour is detected, both API and Native API calls are returned to their original file.

Most approaches that use API calls to detect computer viruses operate at either user level (Win32 APIs) or kernel level (Native APIs). The problem with the former is that some applications can directly call the kernel and avoid using Win32 APIs [48], allowing them to remain undetected, i.e., this approach tends to give false negatives.

On the other hand, the drawback with using kernel level by itself is that unlike system calls, Native APIs are not completely documented like most Unix systems (system calls in most Unix systems are documented) and are almost entirely hidden from view, with only handful of their functions documented in generally accessible publications [49]. This drawback makes the use of Native APIs incomplete and liable to both false negatives and false positives, so that the system is not fully protected.

Therefore, it can be hypothesised that the use of a combined user and kernel level approach provides a better detection system and minimises the rates of false

negatives and false positives. Such a system is able to examine API calls issued in the user mode and if a file is detected as a virus, no further examination is needed. If, however, it is not considered to be a virus, the detection system will examine it at the kernel level by observing its Native API calls.

In order to apply this approach, a parallel execution tool is needed to run user and kernel level detection simultaneously. ITL can do this, handling both sequential and parallel composition [90] and offering user and kernel level detection at the same time. We can also make the Native API calls used at the kernel level adaptable by using ITL formulae and this allows us to add more Native API calls in the future.

Figure 4.13 shows how the system works. At the user level, API calls are extracted and then sent to AnaTempura, which examines them to see if they match the five categories of steps in a virus behaviour.

However, if the five categories are not detected in the user level API calls, Tempura examines the Native API calls coming from the kernel level. This comparison is similar to the above, but concerns only those categories which have not been detected. For example, if three of the five are discovered in the first comparison, the second one considers only the undiscovered categories. Then, if kernel observation completes the set of five categories, Tempura decides that a virus has been detected and the file will not complete execution.

4.5 Virus Behavioural Specification in ITL

We have declared Cat1, Cat2, Cat3, Cat4, and Cat5 which respectively represent the lists of all API function calls for Find to infect, Get Information, Read and/or Copy, Write and/or Delete, and Set Information, as listed in Table 4.9.

We suppose that X represents all API and Native API calls which are received at runtime by Tempura. X will be received as a text representing all API calls issued by a certain PE file. Note that for a given set A, the predicate 'in$A$(X)' holds if $X \in A$.

- The ITL formulae for Category one will be as follow:

$$Ucat1(X) =$$

$$\left(inUsermode\ (X) \vee inKernelmode\ (X)\right) \wedge inCat1(X). \qquad (1)$$

This formula indicates that if one or more API or Native API calls denoted by X issued by a file in the user or kernel level, is in the list of Cat1.

The previous formula will be applicable for all the categories in the user level except Cat3 and Cat4 that represent the read and write categories respectively.

- Therefore, The ITL formulae for Category two, and five will be as follow:

$$Uca2(X) =$$

$$\left(inUsermode\ (X) \vee inKernelmode\ (X)\right) \wedge inCat2(X). \qquad (2)$$

$$Ucat5(X) =$$

$$\left(inUsermode\ (X) \vee inKernelmode\ (X)\right) \wedge inCat5(X). \qquad (3)$$

However, several rules and conditions should be considered in this research. Firstly, in order to write to an existing or new file, a virus will read and write in order, i.e., will read first and then write to the infected file. Secondly, one of the API calls (*ReadFile, ReadFileEx, OpenFile, OpenFileByld, and ReopenFile*) must be called in the third category and one of the API calls (*WriteFile, WriteFileEx and ReplaceFile*) be called in the fourth category.

- Therefore the formula of Category three will be as follow

$$Ucat3(X) =$$

$$\left(inUsermode(X) \vee inKernelmode\ (X)\right) \wedge\ inCat3(X) \wedge\ inRead(X). \qquad (4)$$

Where *Read* = (ReadFile, ReadFileW, OpenFile, OpenFileByld, ReopenFile).

- The formula for Category four will be

$$Ucat4(X) =$$

$$\left(\ inUsermode(X) \vee inKernelmode\ (X)\right) \wedge\ inCat4(X) \wedge\ inWrite(X). \qquad (5)$$

Where *Write* = (WriteFile, WriteFileEx and ReplaceFile).

- Because the order of read and write is very important in this research, the next formula will be applicable:

$$\Diamond Ucat3(X)\ ;\ \Diamond Ucat4(X). \qquad (6)$$

It shows that the write calls must be issued sometimes (◊) after a read call. However, if one or more categories are not detected at the user level, then their Native API calls coming from the kernel will be examined.

- Therefore the next formula will be used:

$$\Box\,(\neg inUsermode(X) \equiv inKernelmode(X) \wedge (Ucat1(X) \vee Ucat2(X) \vee Ucat3(X) \vee Ucat4(X) \vee Ucat5(X)) \quad (7)$$

The previous formula indicates that if one or more categories have not been detected in the user level, they will have more examination at kernel level, in order to see if there is a call belongs to the undetected category that has been directly issued to the kernel. The same mechanism will be used to examine the Native API calls coming from the kernel.

In addition, the order of these API calls is highly significant and indeed the main contribution in this research.

- The following formulae represent the different order of API calls that a virus normally carries out in order to attach itself to another file.

$$\Diamond Ucat1(X)\ ;\ \Diamond Ucat2(X)\ ;\Diamond Ucat3(X)\ ;\ \Diamond Ucat4(X)\ ;\ \Diamond Ucat5(X). \quad (8)$$

$$\Diamond Ucat1(X)\ ;\ \Diamond Ucat2(X)\ ;\Diamond Ucat3(X)\ ;\ \Diamond Ucat5(X)\ ;\ \Diamond Ucat4(X). \quad (9)$$

$$\Diamond Ucat2(X)\ ;\ \Diamond Ucat1(X)\ ;\Diamond Ucat3(X)\ ;\ \Diamond Ucat4(X)\ ;\ \Diamond Ucat5(X). \quad (10)$$

$$\Diamond Ucat2(X) ; \Diamond Ucat1(X) ; \Diamond Ucat3(X) ; \Diamond Ucat5(X) ; \Diamond Ucat4(X). \quad (11)$$

Previous four formulae illustrate that in order to attach itself to another file, a virus will issue API or Native API calls from the five categories in four orders. Firstly, the normal order from category one to category five. Secondly, a virus will issue calls from category one to three and then five and finally from the fourth category. The third scenario is that a virus will firstly issue a call from category two then category one and subsequently issue calls from category three, four, and five respectively. The final scenario is that a virus initially issues a call from category two then category one and succeeding that category three, five, and four sequentially. Indeed, these orders need to be considered besides the rule associated with the read category (third category). Moreover, these different scenarios will be discussed further in Chapter 5.

4.6 Summary

This chapter has set out and discussed the architectures of both virus behaviour and detection in order to identify how a virus can infect a system and how it can be detected. It has considered how to determine the behaviours of viruses, using a number of tools, and discussed the methods used to gather and analyse data on computer viruses. Examining API calls was found to be one of the best methods to determine the behaviour of viruses and to trace them. This research has identified five categories of steps that can occur in a virus behaviour as it infects a system, starting with finding a file or a directory to infect. Next, it acquires information such as its attributes and directory path, then reads and/or copies a file which has the virus in

order to write in the subsequent behaviour. The fifth and final category observed was that of setting information in a file in order to change its attributes for future use. User level API calls known as Win32 APIs were traced to determine whether a malicious action had occurred. To achieve a better result, this method was supplemented by observing kernel level API calls (Native APIs), which can be used to call the kernel directly, avoiding the Win32 APIs.

The method used to detect the behaviour of computer viruses by tracing their API and Native API calls was discussed and its architecture drawn at both user and kernel level. If the five categories of behaviour discussed above are observed at the user level, then a virus has been detected; otherwise the kernel Native API calls are examined to determine whether any undetected categories can be found at that level. If all five categories are discovered at the user or kernel level or both, then a virus is present; otherwise, the file is considered benign. In addition, ITL formulae which represent virus behaviour were provided at the end of this chapter.

ITL's executable subset, Tempura will be examined in the next chapter in order to demonstrate its value to this research in detecting virus behaviour. Tempura and other tools that are plugged-in with it will be discussed in detail in the next chapter. In addition, the next chapter will discuss the implementation, how the architecture works, and how viruses are detected in detail.

Chapter 5

AnaTempura Integration and Implementation

Objectives:

- Develop a prototype implementation of behaviour-based virus detection.
- Discuss AnaTempura and the tools that are needed to be plugged-in with it.
- Provide the functions that are used to detect computer viruses.
- Describe the algorithm that is used by Tempura to detect computer viruses.

5.1 Introduction

This chapter explains how Tempura can be used to detect computer viruses by means of their API calls. It also explains which tools should be used to extract Win32 and Native API calls that represent the steps of virus behaviour at both user and kernel levels.

Tempura will examine both Win32 and Native API calls at run time. As explained previously, five stages have been identified in which a virus operates in order to attach itself to another file or files. First, it finds a place to infect, which is normally the file system of a Windows operating system. Second, it obtains the information of the wanted file or directory. Next, it reads and/or copies the file in which it should first read itself before writing to another place.

Fourth, to attach itself to another file, a virus must write after it reads itself and it mostly writes to one of four places: a virus may replace its content with an existing file by overwriting it (Parasitic), it may attach itself to an existing file by injecting its code into this file and replacing its entry points (Overwriting), it may change the name of an existing file with its original name (Companion), or it may create a new file and write to this file. Finally, Set File Information category's API calls will also be called to change the file attributes which can also be used in the future as a channel for the hacker to infect the system again, or to use the same file to read and write in the future.

As mentioned previously, examining only Win32 API calls will not be enough because some viruses exist which directly contact the kernel ending with a false

positive and false negative. Therefore, both Win32 and Native API calls should be examined by Tempura. This means that both the kernel and user level calls should be examined at the same time. This chapter explains the implementation of our prototype at both kernel and user levels, besides the tools that help Tempura to detect computer viruses at both levels.

5.2 What Is Needed

At the user level, information of all wanted API calls should be monitored alongside its process ID (PID), the file which issues the call, and the parameters that are needed in order to complete the call. On the other hand, the kernel level is responsible for intercepting a number of selected Native API calls and then going all the way back to Tempura, telling it if one or more of these Native API calls has been issued. This is shown in Figure 5.1.

The reason why we choose to hook only the wanted API and Native API calls at both levels is related to the performance of the machine. Marhusin et al. [91] suggested minimising the number of hooks to achieve the hooking goal while preserving reasonable computer performance [91]. Therefore, by observing all the needed API calls at the user level and intercepting only the selected Native API calls at the kernel level, the performance of the system will be better than if all calls at both levels are observed.

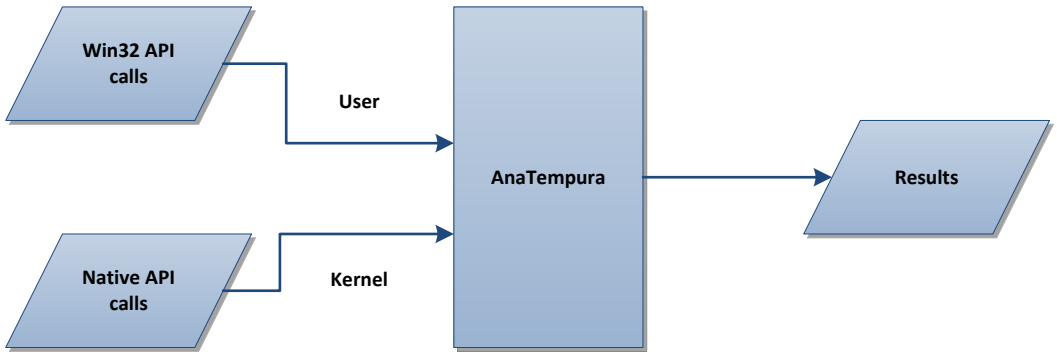


Figure 5.1: General Integration with AnaTempura .

Practically, it is better to start with the user level to see how the incoming Win32 API calls can be compared with the analysed categories and how many API calls a virus can issue during runtime at the user level. As a result, each call will be compared with each category and the outcome of this comparison will determine how many steps this particular file has carried out.

5.3 User Level

Figure 5.1 shows that, a tool which works at the user level is needed to monitor selected API calls, and provide the parameters associated with these calls, in order to fulfil the requirements of this research. This tool should be able to be plugged-in with Tempura that is a plug-in tool by its nature. This tool should provide its output to Tempura in order to complete the detection mechanism. Deviare API [92] has been found as the most practical tool to deal with API calls at the user level as explained in the next section.

5.3.1 Deviare API

The prototype of this research has integrated a tool called Nektra's Deviare API [92] for intercepting Windows API calls on the fly. Deviare API provides hook libraries to intercept any Windows API calls at runtime. It is an API hook engine that was designed to create end-user products.

Intercepting API calls is not an easy task which can take place in many different scenarios. Our choice of Deviare API is due to the fact that the developers of this tool have tested all the different scenarios to avoid unpleasant crashes that might cause damage to the system. Most popular hook engines do not address these issues [92]. Although they work in many situations, a truly professional hook engine that works in all situations is needed to fulfil the objective of this research.

As well as this, Deviare API is a generic API interception engine. Its main difference from other hook engines is that it allows us to intercept different functions with a single handler and decide which functions to hook at runtime. To intercept APIs with any other product, a specific handler that runs in the target process context has to be written. In contrast, one of the advantages that the Deviare API provides a generic handler which has the ability to receive all API calls by each process.

Deviare API provides a COM interface supported by most programming languages such as C++, Delphi, VB, VB.SCRIPT, VB.NET, C-Sharp (C#) and Python. API hooking can be used in different fields such as learning about the internal behaviour of the Windows operating system and the behaviour of the external applications, without access to their source code, malware analysis, tracing and debugging an application's

code execution, showing its API calls and parameters. The latter advantage of API hooking is the one that can be provided by Deviare API which will help us with identifying a set of API calls with their parameters in order to precisely examine a process.

As shown in Figure 5.2, Deviare API provides the ability to receive the API call as well as the process that issues the call, the Process ID and the parameters associated with this API call. Therefore, by receiving all this information at the user level from Deviare API, AnaTempura will be able to examine and compare them with the five categories observed previously in Section 4.3.

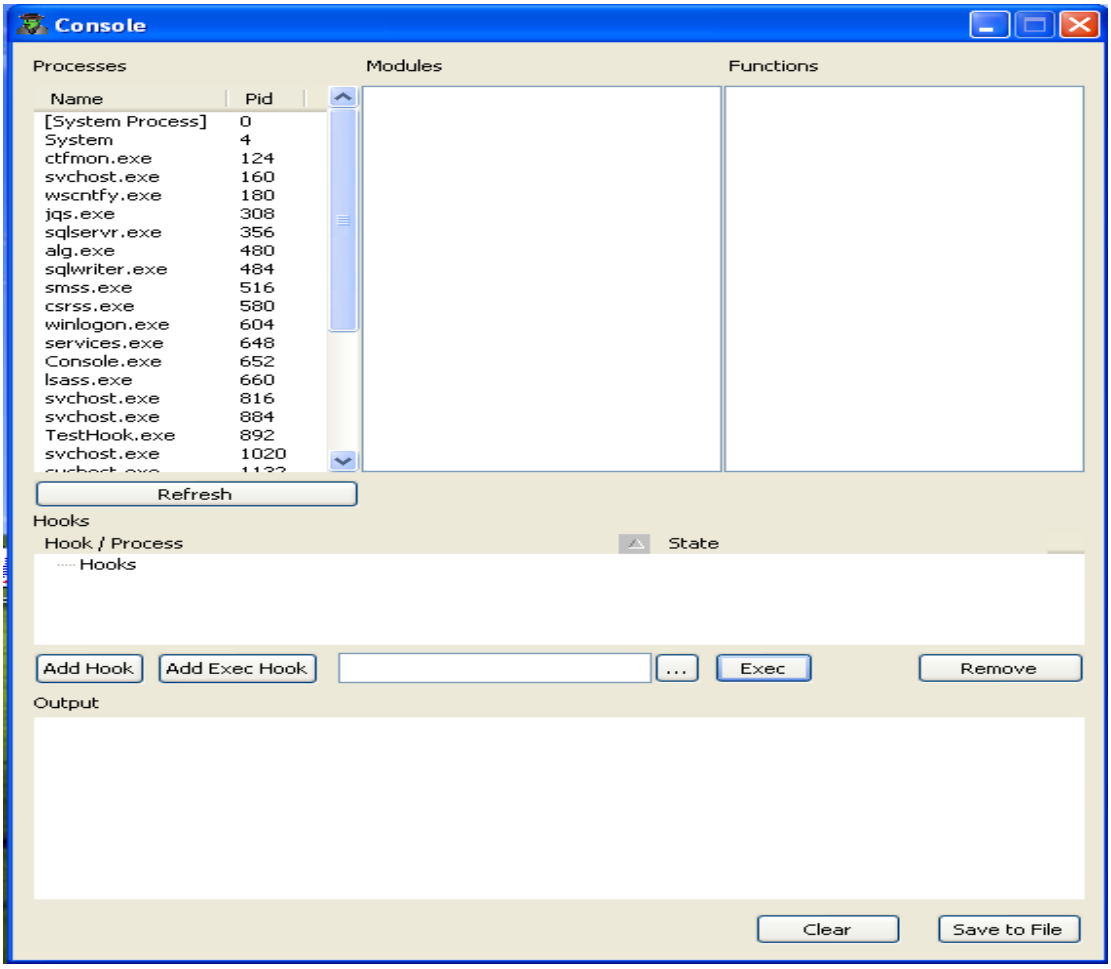


Figure 5.2: Deviare API commercial edition.

Not all API calls' parameters are needed. As observed by this research, some API calls are obvious and there is no need to examine their parameters. API calls such as those which belong to the read category are in fact essential to see what the file has read. In addition, parameters associated with the API call that creates a file might also be needed for classification to see where the file has been created and whether it creates a new file or overwrites an existing one.

Table 5.1: dwCreationDisposition parameter.

Value	Meaning
CREATE_ALWAYS 2	Creates a new file, always. If the specified file exists and is writable, the function overwrites the file, the function succeeds, and the last-error code is set to **ERROR_ALREADY_EXISTS** (183). If the specified file does not exist and is a valid path, a new file is created, the function succeeds, and the last-error code is set to zero.
CREATE_NEW 1	Creates a new file, only if it does not already exist. If the specified file exists, the function fails and the last-error code is set to **ERROR_FILE_EXISTS** (80). If the specified file does not exist and is a valid path to a writable location, a new file is created.
OPEN_ALWAYS 4	Opens a file, always. If the specified file exists, the function succeeds and the last-error code is set to **ERROR_ALREADY_EXISTS** (183). If the specified file does not exist and is a valid path to a writable location, the function creates a file and the last-error code is set to zero.
OPEN_EXISTING 3	Opens a file or device, only if it exists. If the specified file or device does not exist, the function fails and the last-error code is set to **ERROR_FILE_NOT_FOUND** (2).
TRUNCATE_EXISTING	Opens a file and truncates it so that its size is zero bytes,

5	only if it exists. If the specified file does not exist, the function fails and the last-error code is set to **ERROR_FILE_NOT_FOUND** (2). The calling process must open the file with the **GENERIC_WRITE** bit set as part of the *dwDesiredAccess* parameter.

For example, the parameter 'dwCreationDisposition' of *CreateFile*, as mentioned by [93], is an action to take on a file or device that exists or does not exist. This parameter must be one of the following values, which cannot be combined, as shown in Table 5.1. Therefore, by receiving and observing this parameter, we can classify the virus after is has been caught if it is writing to a new place or to an existing one.

As shown in Table 5.2, there are also significant parameters in the read *ReadFile* API call. The most significant is a handle to the file which has been read and this parameter cannot be neglected because it helps to determine whether a file is a virus or not in order to see whether a file has read itself or not.

Table 5.2: *ReadFile* and *CreateFile* Prototype.

API call	parameters

ReadFile	`BOOL WINAPI ReadFile(`
	`  __in  HANDLE hFile,`
	`  __out  LPVOID lpBuffer,`
	`  __in  DWORD nNumberOfBytesToRead,`
	`  __out_opt  LPDWORD lpNumberOfBytesRead,`
	`  __inout_opt  LPOVERLAPPED lpOverlapped`
	`);`

CreateFile	HANDLE WINAPI CreateFile(__in LPCTSTR lpFileName, __in DWORD dwDesiredAccess, __in DWORD dwShareMode, __in_opt LPSECURITY_ATTRIBUTES lpSecurityAttributes, __in DWORD dwCreationDisposition, __in DWORD dwFlagsAndAttributes, __in_opt HANDLE hTemplateFile);

5.3.2 API Calls Intercepting and Parameters

To start a new Windows Application project using C# that works simultaneously with Deviare API, a number of library references need to be added to the project in order to complete the experiment. For example, those which are responsible for getting the API call's parameters are in fact significant for examining these parameters associated with each call. Thus, all Deviare library references have been added to this project.

In order to use Deviare API, its namespace as shown in the code below needs to be included. By including this name space, all its intercepting features can be used and it can also enable the project to deal directly with the API calls.

Listing 5.1: Deviare API namespace.

```
using Nektra.Deviare2;
```

When the *OnFunctionCalled* event is received, three objects will be received as parameters as shown in Listing 5.2. First, an *INktHook* indicating which hook fired the event. Second, an *INktProcess* object where the process Id and name using the "Id" and "Path/Name" properties can be extracted. Third, an *INktHookCallInfo* object that provides specific information about the call. One of them is the "Params" method which in turn will return a collection of parameters. With that collection each

INktParam object can be used to access the type of the parameters, inspect and/or change its value, etc.

Listing 5.2: OnFunctionCalled Function.

```
private void OnFunctionCalled(NktHook hook, NktProcess process,
NktHookCallInfo hookCallInfo)
```

The following code shows how *CreateFileW*[1] system calls can be intercepted. This API call has been randomly chosen due to the fact that explaining one API call is the same as explaining other API call intercepting. In addition, it has more parameters that are significant for classifying the virus.

Listing 5.3: *CreateFile* Intercept.

```
 NktHook hook = _spyMgr.CreateHook("kernel32.dll!CreateFileW",
(int)(eNktHookFlags.flgRestrictAutoHookToSameExecutable &
eNktHookFlags.flgOnlyPreCall));
    hook.Hook(true);
    hook.Attach(_process, true);
```

The subsequent snippet shows how the parameters of an API call can be intercepted. Note, the *CreateFile* API call prototype has been shown in Table 5.2.

Listing 5.4: Call Information.

```
 INktParamsEnum paramsEnum = hookCallInfo.Params();
```

The codes below show how the seven parameters of the *CreateFile* API call which are shown in Table 5.2, can be observed using the Deviare API tool. Since the type of each parameter is previously known, it can be easier to obtain these

[1] CreateFileA is for **ANSI** format and CreateFileW is for **Unicode** format. Therefore, it is up to the system in which if the Unicode format is used, the CreateFileW will be called and vice versa.

parameters. These parameters can also be formatted in a way that Tempura can handle them.

Listing 5.5: lpFileName parameter.

```
//lpFileName
    INktParam param = paramsEnum.First();
    strCreateFile += param.ReadString() + "\", ";
```

Every parameter can be interpreted and translated into a symbolic constant. For example, the following piece of code shows how the created file can be accessed.

Listing 5.6: dwDesiredAccess parameter.

```
//dwDesiredAccess
    param = paramsEnum.Next();
       if ((param.LongVal & 0x80000000) == 0x80000000)
        strCreateFile += "GENERIC_READ ";
         else if ((param.LongVal & 0x40000000) == 0x40000000)
         strCreateFile += "GENERIC_WRITE ";
          else if ((param.LongVal & 0x20000000) == 0x20000000)
           strCreateFile += "GENERIC_EXECUTE ";
            else if ((param.LongVal & 0x10000000) == 0x10000000)
             strCreateFile += "GENERIC_ALL ";
               else
                strCreateFile += "0";
                 strCreateFile += ", ";
```

The subsequent code shows the share mode of the Created file API. It can be seen that a normal file can be in read, write or delete share mode by changing its attributes.

Listing 5.7: dwShareMode and lpSecurityAttributes parameters.

```
/dwShareMode
   param = paramsEnum.Next();
     if ((param.LongVal & 0x00000001) == 0x00000001)
      strCreateFile += "FILE_SHARE_READ ";
        else if ((param.LongVal & 0x00000002) == 0x00000002)
         strCreateFile += "FILE_SHARE_WRITE ";
          else if ((param.LongVal & 0x00000004) == 0x00000004)
           strCreateFile += "FILE_SHARE_DELETE ";
            else
                strCreateFile += "0";
```

```
                strCreateFile += ", ";

//lpSecurityAttributes
            param = paramsEnum.Next();
            if (param.PointerVal != IntPtr.Zero)
            {
                strCreateFile += "SECURITY_ATTRIBUTES(";

                INktParamsEnum paramsEnumStruct =
param.Evaluate().Fields();
                INktParam paramStruct = paramsEnumStruct.First();

                strCreateFile += paramStruct.LongVal.ToString();
                strCreateFile += ", ";

                paramStruct = paramsEnumStruct.Next();
                strCreateFile += paramStruct.PointerVal.ToString();
                strCreateFile += ", ";

                paramStruct = paramsEnumStruct.Next();
                strCreateFile += paramStruct.LongVal.ToString();
                strCreateFile += ")";
            }
            else
                strCreateFile += "0";
            strCreateFile += ", ";
```

As shown in Table 5.1, the dwCreationDisposition parameter can help with classifying the virus if a new file has been created or not. Listing 5.8 shows how dwCreationDisposition parameters can be one of the variety of options which were explained in Table 5.2.

Listing 5.8: dwCreationDisposition, dwFlagsAndAttributes and hTemplateFile parameters.

```
//dwCreationDisposition
            param = paramsEnum.Next();
            if (param.LongVal == 1)
                strCreateFile += "CREATE_NEW ";
            else if (param.LongVal == 2)
                strCreateFile += "CREATE_ALWAYS ";
            else if (param.LongVal == 3)
                strCreateFile += "OPEN_EXISTING ";
            else if (param.LongVal == 4)
                strCreateFile += "OPEN_ALWAYS ";
            else if (param.LongVal == 5)
                strCreateFile += "TRUNCATE_EXISTING ";
            else
                strCreateFile += "0";
            strCreateFile += ", ";
//dwFlagsAndAttributes
            strCreateFile += param.LongVal;
```

```
            strCreateFile += ", ";
//hTemplateFile
            strCreateFile += param.LongLongVal;
            strCreateFile += ");\r\n";
```

Note that the attributes associated with the *CreateFile* API call were tested in this research for classifying purpose. In other words, some computer viruses were tested to observe whether they attach themselves to new or existing files in order to show our prototype's ability in classifying these viruses. However, the classification is not part of the final prototype due to the fact that the prototype's main role is only to detect computer viruses and distinguish between them and benign processes. Therefore, the *CreateFile* API's parameters were tested in this research to show that more information about computer viruses can be acquired from API calls.

Owing to the fact that the process ID and process name are considered as significant factors in this research, they have to be specified with each system call, i.e., all the information about each API call should be given with the process that issues the call and its identifier. The code below shows how the process name and the process ID can be obtained.

Listing 5.9: Process Name and ID.

```
//ProcessId

            process.Id.ToString()
//Process Name

            process.Name
```

This means that, the snippets above show that Deviare API with C# are able to provide three main parameters, namely: The API call that is running, parameters

associated with this call, the process ID that issues the call. The fourth parameter is the file name that is read by the process and this can be acquired as shown in Listing 5.10.

Listing 5.10: Filename parameter.

```
if (hook.FunctionName == "kernel32.dll!ReadFile" || hook.FunctionName
== "kernel32.dll!ReadFileEx"
                    || hook.FunctionName == "ntdll.dll!NtReadFile")
                {
                    IntPtr h = paramsEnum.GetAt(0).SSizeTVal;
                    string s = "";
                    if (h != IntPtr.Zero)
                    {
                        try
                        {
                            s = _tool.GetFileNameFromHandle(h,
process);
                        }
                        catch (Exception e)
                        {
                            throw (e);
                        }
                    }

                    int i = s.LastIndexOf('\\');
                    s = s.Substring(i + 1);

                    strCreateFile += s;
                    strCreateFilea += s;
                }
```

5.3.3 Deviare API Output

The output of Deviare API will be read by Tempura using assertion points (explained in detail in Appendix B) in the C# program as shown in Listing 5.11.

Listing 5.11: Assertion Points.

```
{
//Assertion Points
Console.WriteLine("!PROG: assert
Name:{0}:Id:{1}:Call:{2}:Inputs:{3}:!", P_name, P_ID, Call, hfile);

}
```

However, the C# program will be wrapped in a Java program to enable us to make the output readable by Tempura. In other words, the Java program acts as a pipe between Tempura and the C# program. As a result, the output of the combination between C# and Deviare API will appear as follows:

Listing 5.12: C# Output which are read by Tempura.

```
!PROG: assert Name:Eyeveg.exe:Id:2331:Call:CreateFileW:Inputs::!
!PROG: assert
Name:Eyeveg.exe:Id:2331:Call:NtQueryDirectoryFile:Inputs::!
!PROG: assert Name:chrome.exe:Id:6092:Call:NtClose:Inputs::!
!PROG: assert Name:chrome.exe:Id:6092:Call:CloseHandle:Inputs::!
!PROG: assert
Name:Eyeveg.exe:Id:2331:Call:ReadFile:Inputs:kernel32.dll:!
!PROG: assert
Name:Eyeveg.exe:Id:2331:Call:NtSetInformationFile:Inputs::!
!PROG: assert
Name:Eyeveg.exe:Id:2331:Call:NtSetInformationFile:Inputs::!
!PROG: assert
Name:Eyeveg.exe:Id:2331:Call:NtSetInformationFile:Inputs::!
!PROG: assert Name: firefox.exe:Id:4284:Call:NtClose:Inputs::!
!PROG: assert Name: firefox.exe:Id:4284:Call:CloseHandle:Inputs::!
```

5.3.4 Deviare API & Tempura

As abovementioned, Deviare API can be plugged in with most programming languages. Therefore, in this implementation, the C# programming language has been used to retrieve the API calls from the Deviare API engine with the process which issues the calls, its process ID and the parameters associated with this call. Subsequently, Tempura will be able to receive all the data from Deviare API and then examine them. Therefore, the final implementation of Deviare API, C#, and Tempura at the user level will appear as shown in Figure 5.3.

Tempura has the ability to be plugged-in with most programming languages such as Java, C, etc. Therefore, the Java program will give the information from the C# program to Tempura.

Tempura will receive the four parameters explained previously from Deviare API as separate lists. Each list should be provided as a string list in order to be captured by Tempura. Each time the Deviare API runs, it will directly give all the information to

the Java program that acts as a pipe and will be read at the same time by Tempura as shown in the figure below. Then, Tempura will deal with these lists that come from the user level and provide a result if the five categories have met with the conditions associated with these categories. The following subsections will explain the functions that have been used in Tempura at the user level.

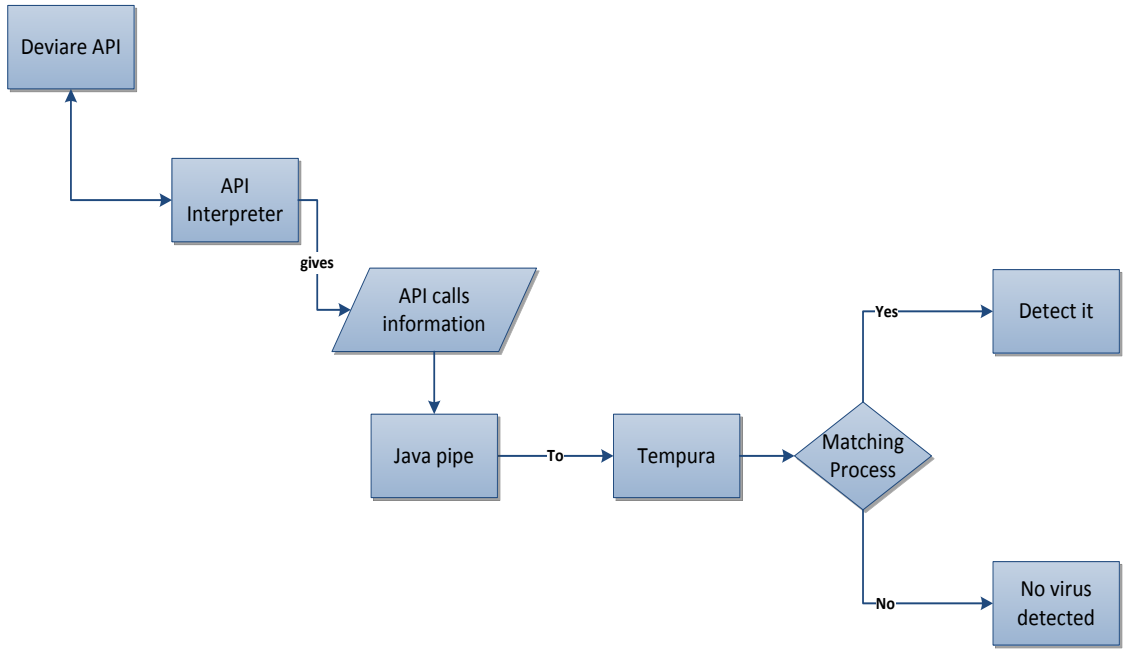


Figure 5.3: Deviare API, C# and Tempura.

5.3.5 User Level Tempura Functions

Several functions at the user level have been defined in Tempura to check whether the observed file is a virus or not. Firstly, seven lists*: Cat1, Cat2, Cat3, Cat3A, Cat4, Cat4A* and *Cat5* have been declared. They represent finding a place to infect, getting the

information, reading and/or copying, writing and/or deleting, and setting file information categories respectively. These functions are responsible for examining the API calls at the user Level and are detailed in the following bullet points.

- **Process ID**

There are a huge number of API calls that can be monitored at run-time. They can be issued by different concurrent running processes at the same time. Virtually, every process running in a system, issues system calls, however they are not mixed and can easily be differentiated for every process [45]. Therefore, every call is associated with a unique process identifier (PID) in order that calls coming from different processes do not mix. For example, if an API call belongs to category one with PID 1674, it will not mix or be linked with an API call from category two which is associated with 1624 PID. However, the Process ID can help with identifying the process name which will be explained in the next bullet point.

- **Process Name**

As shown in Figure 5.4, the process name can be inferred from its associated process ID. This figure has been captured to displays the process name and ID, and other information such as, local port, protocol and local port name [94].

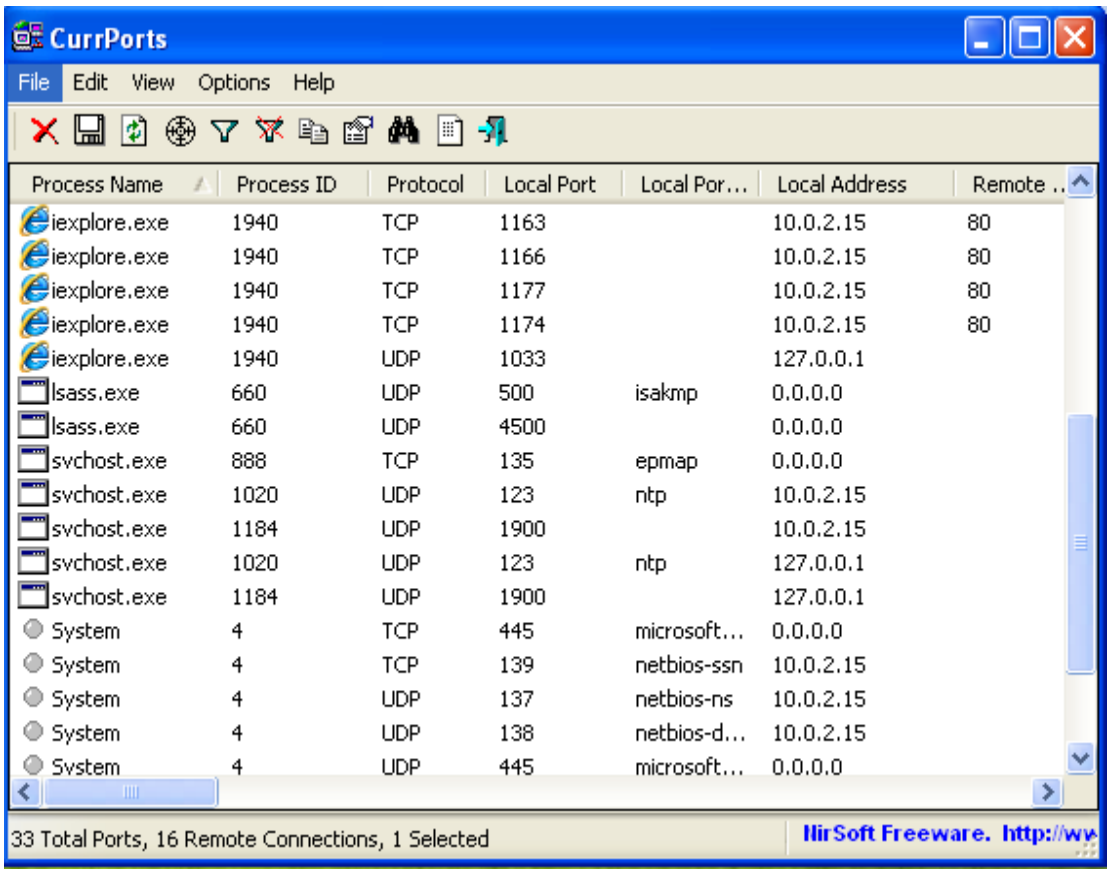


Figure 5.4: Process name and ID.

The process name is particularly important in order to be used as a comparison when a file is read, as explained previously. The process name will be compared in case a read call has been issued to see if the read file matches with the process that is reading this file. This means that, if the file has read itself, then it can be considered to be a possible virus if it satisfies all the mentioned conditions.

- **Inputs (API Parameters)**

Each API call carries with it a number of inputs or parameters that are required to complete the execution of the call. These parameters can be a handle to read a file, as shown in the code below, to set security privileges, and other inputs that are essential to finish the call.

Listing 5.13: ReadFile API call Code.

```
BOOL WINAPI ReadFile(
  __in          HANDLE hFile,
  __out         LPVOID lpBuffer,
  __in          DWORD nNumberOfBytesToRead,
  __out_opt     LPDWORD lpNumberOfBytesRead,
  __inout_opt   LPOVERLAPPED lpOverlapped
);
```

The *ReadFile* windows API call is present here as an example because it will be one of the most important Win32 API call, the parameter of which will be investigated in depth. As mentioned previously, in order to attach itself to another file and to be distinguished from other normal processes, a virus must read itself. Therefore, the name of the file that is read by the *ReadFile* will be examined each time this API call is issued.

Then, the name of the process that issues the *ReadFile* API will be compared with the read file to see if there is a match. Other API calls' parameters that belong to the read category will also be important in order to see the read file, such as, *OpenFile and ReopenFile* as shown below in Listing 5.14. The prototypes of the latter API calls are shown in the code below.

Listing 5.14: OpenFile and ReOpenFile API call Prototypes.

```
HFILE WINAPI OpenFile(
  _In_   LPCSTR lpFileName,
  _Out_  LPOFSTRUCT lpReOpenBuff,
  _In_   UINT uStyle
);
```

```
HANDLE WINAPI ReOpenFile(
  _In_  HANDLE hOriginalFile,
  _In_  DWORD dwDesiredAccess,
  _In_  DWORD dwShareMode,
  _In_  DWORD dwFlags
);
```

- **Check for API Calls**

The *CheckApICalls* function is the main function that will be used to check for API calls that come from both the user Level (Win32 APIs) and kernel level (discussed later). It should deal with four inputs: the Win32 API call(s), the process ID, the process name and the parameters coming with the call. When a call is issued by a particular process at the user level, it will be examined by Tempura to test whether it belongs to one of the five categories explained above. As shown below in Listing 5.15, *CheckApICalls* receives five significant parameters. They are *S*, the process name, the process ID, the call, and the inputs or parameters.

Listing 5.15: CheckApiCalls function.

```
Define   CheckApiCalls(N,S,ProcessName, ProcessId, Call,Inputs)
```

If a call belongs to category one or two then *S* will be assigned to 1 (category one) or 7 (category two) as shown in Figure 5.5. If these first two categories have been issued then, the remaining categories can be examined; otherwise, *S* will be reassigned to 0 or remain as it is. Therefore, if a process issues a call that belongs to category

three, then the *CheckApICalls* function will firstly look in *S* if it is equals to 3 or 9 as shown below in Figure 5.5.

Category three has been split into two lists, one for read which must be issued by a virus and what is called the copy list is not significant, i.e., the copy list will be optional for a virus to issue. The reason why this category has been divided into two lists is because calls such as copy and create are not considered to be malicious calls, however they can assist with classifying the virus in the future. In other words, if a file has issued calls that belong to all the categories and satisfies all the rules considered in this research, then this file can be considered to be a virus and, in addition, it can be classified. For example, if the *CreateFile* API call has been issued from a file that is considered to be a virus, it can be known that this virus is going to write itself to a new file by examining its parameters.

However, read and/or copy is the most important category that distinguishes between normal and malicious action in the system. The process name and the inputs will play a consequential role in investigating a process. This means that, if a call belongs to the read category, then the process name and the input arguments that this call requires will be examined. Thus, if the process name matches the file which has been read, then it can be considered to be a candidate virus and this is the idea behind attachment [45]. In this case, the *S* variable will be assigned to 3 or 9 indicating that this file can be considered as a candidate virus if calls have been issued from the former categories (Category one & two).

Therefore, if a file is considered to be a candidate virus, its attributes have been changed and this file has issued a number of calls that belong to the five categories and satisfies the rules, then a decision can be made.

As the order of the last two categories is not significant in determining whether a malicious action has occurred, they can be issued at any time after a read call has been issued. As a result, if a call belongs to the fourth category, write and/or delete, the *CheckApICalls* will check whether it is a candidate virus and it will also check if a call has been issued from the last category as shown in Figure 5.5.

The write category is also divided into two lists in order to classify the virus if it happens in the future. The first list is the write list which must be issued and the other is the delete list that is optional and they do not play a great role in determining a virus. As a result, if a file is considered to be a virus and it is found that it issued calls that belong to the delete list then it can be considered to be a destructive virus.

The last category is set information or the changing attributes category which is essential for the virus to spread that gives itself the ability to write by enabling the write privilege of the file. In this category, the *CheckApICalls* will also check whether there is a call that belongs to the previous category (Category four) and it will also check whether the file is a candidate virus. If it is, then the *CheckApICalls* function can decide that there is a sequence of calls that can be considered to be a malicious action and that a virus is detected.

Furthermore, the following formulae represent the different order of API calls that a virus normally carries out in order to attach itself to another file. These formulae

were discussed in Chapter 4 and present here to discuss the different order of API calls in further details.

$$\Diamond Ucat1(X)\ ;\ \Diamond Ucat2(X)\ ;\Diamond Ucat3(X)\ ;\ \Diamond Ucat4(X)\ ;\ \Diamond Ucat5(X). \quad (8)$$

$$\Diamond Ucat1(X)\ ;\ \Diamond Ucat2(X)\ ;\Diamond Ucat3(X)\ ;\ \Diamond Ucat5(X)\ ;\ \Diamond Ucat4(X). \quad (9)$$

$$\Diamond Ucat2(X)\ ;\ \Diamond Ucat1(X)\ ;\Diamond Ucat3(X)\ ;\ \Diamond Ucat4(X)\ ;\ \Diamond Ucat5(X). \quad (10)$$

$$\Diamond Ucat2(X)\ ;\ \Diamond Ucat1(X)\ ;\Diamond Ucat3(X)\ ;\ \Diamond Ucat5(X)\ ;\ \Diamond Ucat4(X). \quad (11)$$

These four formulae illustrate that in order to attach itself to another file, a virus will issue API or Native API calls from the five categories in four orders, as shown in Figure 5.5. Firstly, the normal order from category one to category five (S1, S2, S3, S4, and S5). Secondly, a virus will issue calls from category one to three and then five and finally from the fourth category (S1, S2, S3, S6, and S5). The third scenario is that a virus will firstly issue a call from category two then category one and subsequently issue calls from category three, four, and five respectively (S7, S8, S9, S10, and S11). The final scenario is that a virus initially issues a call from category two then category one and succeeding that category three, five, and four sequentially (S7, S8, S9, S12, and S11), as shown in Figure 5.5. In fact, these orders need to be considered besides the rules associated with the read category (third category). Therefore, whenever a file issue calls from the five categories with their rules, one of their orders (that is, *S* reaches 5 or 11), this file will be detected and considers to be a computer virus.

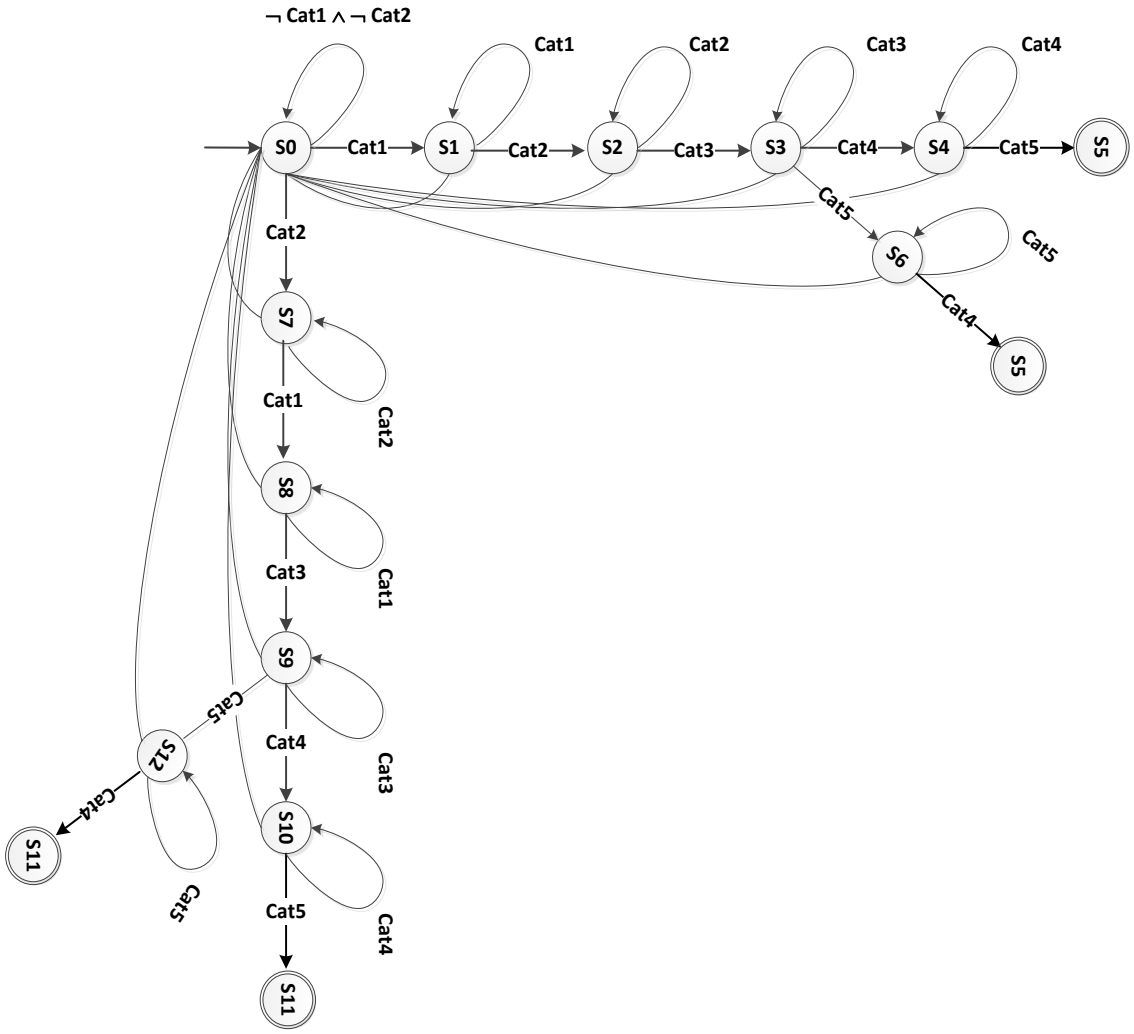


Figure 5.5: Sequences of Virus API calls.

- **Candidate Virus**

A file is considered to be a possible virus if it carries out several steps. Firstly, if calls match the first two categories (S=1 & S=2 or S=7 & S=8) as shown in Listing 5.16. Then, if it requests (a) call(s) that belong/s to the third category and it also obey/s the rules associated with this category (S=3 or S=9). Category 3 will only be significant if API calls issued belong to the previous and subsequent categories.

Listing 5.16: Category one & two (S=1 & S=2).

```
if S=0 and (exists i<|cat1|:
                        cat1[i]=Call)
          then {
                        S:=1 and
         format(" PID:%s issued a call in cat1 \n", ProcessId)
      }
```

```
if S=1 and (exists i<|cat2|:
                        cat2[i]=Call)
          then {
                        S:=2  and
          format(" PID:%s issued a call in cat2 \n", ProcessId)
        }
```

Therefore, if the file is considered to be a candidate virus, then it will be ready to be called from the following steps: namely, the write and set information categories. In other words, the last categories can be examined if a file considered as a candidate virus to see if the file has issued a sequence of API calls that belong to each category with the rules associated with each category.

Listing 5.17: Candidate Virus.

```
if S=2 and (exists i<|cat3|:
                        cat3[i]=Call)
          then {
    if (ProcessName=Inputs)
                then {
                 S:=3  and
                format(" The file ( (%s) ) issued calls in the previous two categories
\n",   ProcessName) and
```

```
                format(" And a call issued in cat3 \n")
                and
               format("_____Then_______(it's a candidate virus)__________ \n\n")
                    }
```

- **Categories Sequences**

In order to infect other files, a virus will issue a sequence of API calls that can be from different orders as explained previously in Section 4.5. Firstly, a virus will issue a call from either category one (S=1) or category two (S=7). Subsequently, *S* can be assigned to 2 or 8 depending on the previous calls as shown in Figure 5.5. If the first two categories were issued, then Tempura would decide whether a file is a possible virus (S=3 or S=9) or not.

For example, when a write (S=4 or S=10) or set information API call (S=6 or S=12) has been issued, the process will be checked whether it is a candidate virus or not (S=3 or S=9). Indeed, if the process is a candidate virus, then the previous categories one and two also were issued.

Finally, the file can be considered as a computer virus if it issues API calls from the five categories and follow one of the four order explained previously. As a result, if a file obey one of these orders and *S* reaches 5 or 11, then the file will be considered as a computer virus as shown in Listing 5.18.

Listing 5.18: The file is a computer virus (S=5 or S=11).

```
if (S=5)
        then {
```

```
            S:=5 and
            format(" The file ( (%s) ) is a COMPUTER VIRUS \n", ProcessName)
             }
```

```
if (S=11)
then {
        S:=11 and
      format(" The file ( (%s) ) is a COMPUTER VIRUS \n", ProcessName)
        }
```

- **Category Determination**

Five main lists were defined in Tempura to see which call belongs to which category. Existential quantification is denoted in Tempura by the 'exists' operator. As shown in the Tempura code below, the 'exists' operator needs two arguments in order to complete the checking. They are as follows: the category that needs to be checked if the call belongs to this category, and the call itself.

Listing 5.19: 'exists' operator.

```
(exists i<|cat1|: cat1[i]=Call) /* Find to infect category  */
(exists i<|cat2|: cat2[i]=Call) /* Get information category  */
(exists i<|cat3|: cat3[i]=Call) /* Read category  */
(exists i<|cat4|: cat4[i]=Call) /*  Write category */
(exists i<|cat5|: cat5[i]=Call) /*  Set Information category */
```

As shown in the listing above, the 'exists' operator is used to help our program by checking for the call of the process. This means that, each time a call is issued, it will be checked with other previous calls. Therefore, this operator will help the program to find out whether a sequence of API calls belongs to the same process. In addition, this

operator will be used to check for API and Native API calls in the five categories as shown in Listing 5.19.

- **Sequence Remember**

Due to the fact that Tempura is organised as a sequence of states, variables and lists that might be used in every state, Tempura forgets the value of the variables unless it is specified that they remain stable. In fact, *S* is a helping variable that gives us the ability to keep the sequence of the five categories in order for them to be used in the future.

For example, if a file is considered to be a candidate virus, it needs to be remembered until a write or set information call is issued, otherwise, the *S* variable will be assigned to zero. Therefore, the *S* will help us by assign it to 9 as shown in Listing 5.20.

Listing 5.20: Remember category three (S=9).

```
{
S:=9  and
        format(" The file ( (%s) ) issued calls in the previous two categories \n", ProcessName) and
        format(" And a call issued in cat3 \n")  and
        format("_____Then_______(it's a candidate virus)_____________ \n\n")
                }
```

5.4 Kernel Level

At the kernel Level, selected Native API calls will be intercepted to tell Tempura whether a process has issued some calls that might complete the five categories observed previously at the user level and then be considered to be a virus. By doing that, this approach will prevent any attempt by a virus to directly contact the kernel level.

As shown in Figure 5.1, a combination of the information coming from both the user and kernel level is essential in order to obtain a proper result and a precise final judgment. A rootkit at the kernel level is the one which is needed to redirect the wanted Native APIs to Tempura at the user level.

5.4.1 Rootkit

Due to the fact that this approach is based on a Windows operating system, the kernel level cannot be easily accessed by normal programs [88]. That means that a rootkit is needed to enable some prohibited privileged access to this OS. Once installed, a rootkit provides our approach with the ability to maintain the privileged access for a long time. However, the kernel mode rootkit used in this research can be called a "good rootkit" because it has been built to benefit the system security that is used to detect a virus. Therefore, it can be developed in the future as a device driver to be part of the OS [95].

The rootkit should intercept any Native API calls which were mentioned previously in Table 4.10. These Native API calls are the ones that belong to different

categories. The role of our rootkit is to hand the information of these calls to Tempura at the user level. As [45] detailed, a Native API call can be traced from the kernel to the user level to check its parameters and the process which issues this call. Therefore, API will be tracked back to its origin getting its process ID, name and other parameters [45].

As illustrated in Figure 5.6, the user level has also played a significant role by adding the information needed by Tempura to the Native API calls in order to specify which process issued the selected Native API and their parameters. As a result, by identifying each PID after tracing back each Native API, these calls will not be mixed and can be easily distinguished and examined by Tempura.

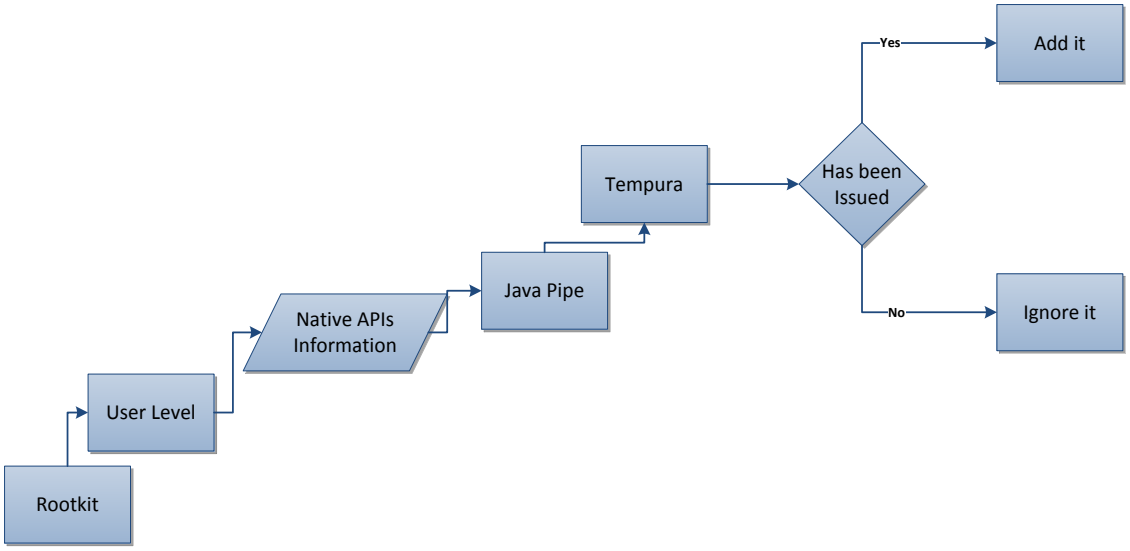


Figure 5.6: Kernel Level and Tempura .

By getting all the needed Native API calls observed by our rootkit and the parameters obtained from the user level, Tempura can ascertain if a direct call has been issued by the same process to the kernel level hence bypassing the user level. Thus, any chance of calling a Native API directly to the kernel will be caught by the

kernel rootkit and then sent to Tempura for further observation. The functions that help Tempura to examine the Native API calls and their attached information coming from the kernel level will be explained in the following section.

5.4.2 Kernel Level Tempura Functions

Tempura kernel level functions that help to examine the incoming Native API calls will be playing their role alongside with the user level functions. Kernel functions should be able to see whether the call made to the kernel has not been done in the user mode. If the call matches an API call at the user level, then this call will be ignored because it is already examined by the user level functions explained previously. These functions will be similar to those which are responsible for the user level API calls. The differences between API and Native API calls' functions are detailed in the following bullet points:

- **Process ID and Name**

The process ID needs to be known by Tempura in order to classify the call, i.e., which process is calling this Native API. Therefore, all Native API calls will not be mixed and can be easily examined. In addition, the process name can be obtained from its PID and it is also important in some API calls such as *NtReadFile and NtOpenFile* in order to tell if the file is reading itself or not.

The Native API calls from a process ID will be matched with their corresponding API calls which have the same PID. As a result, a final judgement will be given by Tempura if these calls coming from the kernel and the user levels satisfy the five categories and their conditions.

- **Inputs (Native API Parameters)**

As explained earlier, not all the parameters are actually needed. There are some clear Native API calls which do not require more examination. In this approach, there are selected Native API calls, three of them need further examination by Tempura at the kernel, namely: *NtOpenFile, NtReadFile* and *NtCreateFile*. These calls require more observation because they can lead to ascertaining whether a file has read itself or not (*NtOpenFile, NtReadFile*) and, also, the file can be classified as to whether it creates a new file or is an existing one (*NtCreateFile*).

- **Check for Native API Calls**

This function will actually be responsible for getting the Native API calls with their parameters, PID and the file name. The role of this function is to examine whether there is a match between a Native API and an API call. For example, if a Process ID has issued a *WriteFile* and, after a while, Tempura has received that the PID has issued a call to the kernel with *NtWriteFile*, the second call will be ignored by this function and thus, will not provide any action.

The idea behind this function is to check every category to see if a previous call has been issued that belongs to this category with the same PID. If so, this call will be ignored and thrown away. Otherwise, and if the call does not belong to any category, a flag will put in the category which it belongs to for future examination.

As mentioned previously, inputs associated with some calls play a significant part in this approach because without them a file cannot be clearly classified. Consequently, this function will also consider these inputs if a selected Native API call

has been issued. For instance, if a Native API call has been issued by a file, this call has not matched with an API call at the user level and it is one of the Native API calls whose parameters are significant such as *NtReadFile* this file will be examined and compared with the read file by this call.

- **Category Determination**

Similar to API calls, Native API calls need also to be examined, i.e., determining which category a Native API call belongs to. Therefore, each time a Native API call is received, it needs to be checked for consistency.

As a result of checking the *S* variable, it can be checked if the coming Native API call has its corresponding API call. This variable will also continue its role by checking for all categories as shown in the code below. For example, if a call has been issued at the user level and a corresponding call has been received from the kernel, no action will be taken (*S* variable will remain as it is) as illustrated in Listing 5.21. On the other hand, if there are no calls that belong to a particular category, this category will be flagged whether the incoming call received is from the user or the kernel (the *S* variable will be assigned another number).

Listing 5.21 : Duplicated calls.

```
if S=1 and (exists i<|cat1|:
                        cat1[i]=Call)
            then {
                        S:=1
                    }
```

The above snippet indicates that if *S=1 (*which means a call from category one was previously issued by a process), and another call is issued from the same category (category one), *S* will remain as it is and Tempura will be waiting for a call from the second category, otherwise *S* will still 1 or return to zero.

5.4.3 Java Pipe

The Java pipe is used in this research to read the information from Deviare API and then transfer them to Tempura. As shown in Listing 5.22, the only role of this code is to run the *TestHook.exe* which work at the same time with Deviare API to intercept calls and delivers them to Tempura that is used simultaneously to examine these calls and decide whether or not an infection has been done. Indeed, this implementation provides only a prototype that is used to demonstrate that API calls can be monitored at runtime with an acceptable performance.

Listing 5.22: Java Pipe Source code.

```
import java.io.*;
import java.util.*;

public class Main1 {
   public static BufferedReader inp;
   public static BufferedWriter out;

   public static void print(String s) {
   System.out.println(s);
   }
   public static String pipe(String msg) {
```

```
String ret;
try {
   ret = inp.readLine();
   return ret;
}
catch (Exception err) {

}
return "really?";
}
public static void main(String[] args) {
String s;
String cmd = "/Users/Sulaiman/Documents/New version/tempura-2.19/Examples/csharp/TestHook.exe";
Console c = System.console();
String in ;
   try {
   Process p = Runtime.getRuntime().exec(cmd);

   inp = new BufferedReader( new InputStreamReader(p.getInputStream()) );
   out = new BufferedWriter( new OutputStreamWriter(p.getOutputStream()) );

            while(inp != null)
            {

   print( inp.readLine());

   };

   inp.close();
```

```
    out.close();
        }

    catch (Exception err) {
    err.printStackTrace();
}
}

        }
```

5.5 Full Approach

The output of both Deviare API and the Kernel rootkit should be combined and examined in Tempura. The full approach of the implementation of this research should be as follows. Two tools are needed at both kernel and user levels, namely the kernel rootkit and Nektra's Deviare API. These two should provide information needed by Tempura to provide comprehensive information about a file. Indeed, it is not easy to program the low level of a Windows operating system because it is not an open source environment. The kernel rootkit should give Tempura the Native API calls alongside their information. However, Deviare API will play the role of a rootkit because it can provide the Native API information coming from the kernel level after the call is accepted. Therefore, the implementation of this research will be as shown in Figure 5.7. At the user level, Tempura will receive a list of API calls with their inputs and process ID and the file which issues this call by means of the Deviare API tool. However, the information coming from the Deviare API at the user level will not be enough and an unreliable result might be given by Tempura.

In order to fill this deficiency, the kernel information has been provided to Tempura to make sure that the decision is as reliable as possible. Deviare API will play the role of the kernel rootkit and deliver this information to Tempura. This means that Tempura will receive the information coming from the user level with that which is delivered from the kernel level and provide a final judgement. It often happens that, with the large amount of information we have here, interference might occur. However, by assigning each call to a unique process these calls will not be mixed even if they come from different sources.

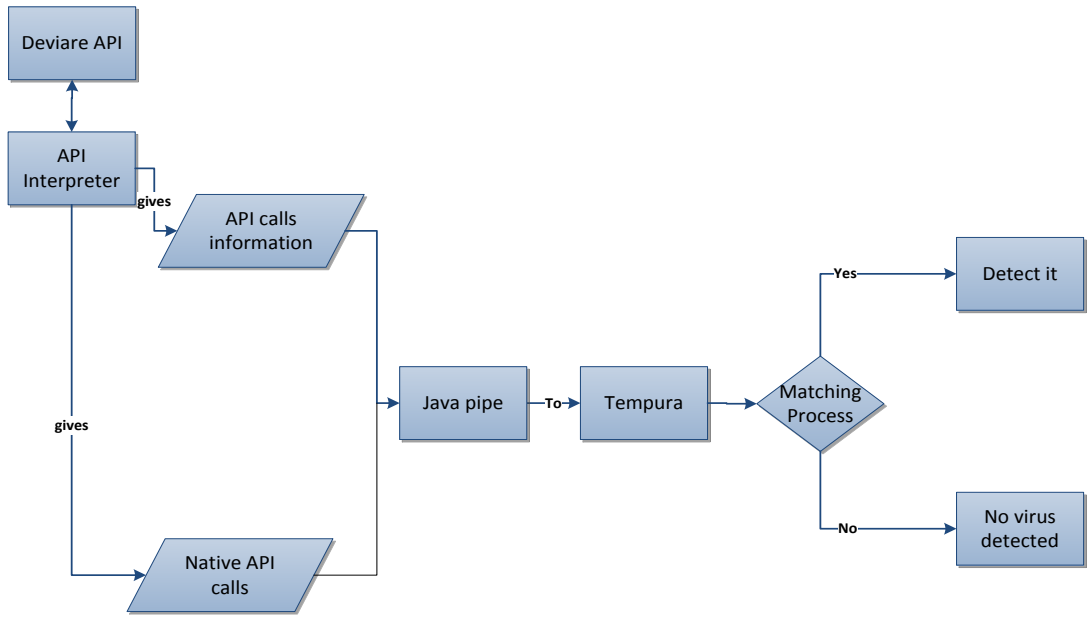


Figure 5.7: Full Approach, and Tempura.

Due to the fact that using both levels to trace API and Native API calls will provide a better result to detect computer viruses, user and kernel levels' information is merged. The combination of both levels has shown that the problem, discussed

formerly in Chapter 4 which is the lack of detection when only one level is used, can be solved. The pseudo code that represents the implementation of Figure 5.5 is shown below in Algorithm 5.1. This pseudo code ensures that the one of the API call orders which computer viruses normally carry out is met.

Algorithm 5.1: Pseudo code of implementation of Figure 5.1.

```
if Pid = i then

S=0   ∧   if S=0 and Call=Cat1    then  S:=1   and
          if S=0 and Call=Cat2    then  S:=7   and
S=1   ∧   if S=1 and Call=Cat1    then  S:=1   and
          if S=1 and Call=Cat2    then  S:=2   and
S=2   ∧   if S=2 and Call=Cat2    then  S:=2   and
          if S=2 and Call=Cat3    then  S:=3   and
S=3   ∧   if S=3 and Call=Cat4    then  S:=4   and
S=4   ∧   if S=4 and Call=Cat4    then  S:=4   and
          if S=4 and Call=Cat5    then  S:=5   and
S=3   ∧   if S=3 and Call=Cat5    then  S:=6   and
S=6   ∧   if S=4 and Call=Cat5    then  S:=6   and
          if S=4 and Call=Cat4    then  S:=5   and
S=5   ∧   if S=5                  then  This file is a virus.]

S=7   ∧   if S=7 and Call=Cat2    then  S:=7   and
          if S=7 and Call=Cat1    then  S:=8   and
S=8   ∧   if S=8 and Call=Cat1    then  S:=8   and
          if S=8 and Call=Cat3    then  S:=9   and
S=9   ∧   if S=9 and Call=Cat4    then  S:=10  and
S=10  ∧   if S=10 and Call=Cat4   then  S:=10  and
          if S=10 and Call=Cat5   then  S:=11  and
S=9   ∧   if S=9 and Call=Cat5    then  S:=12  and
S=12  ∧   if S=10 and Call=Cat5   then  S:=12  and
          if S=10 and Call=Cat4   then  S:=11  and
S=11  ∧   if S=11                 then  This file a virus.]
```

It has been shown by some researches [88, 91] that the performance of the system might be overloaded when both levels are used. However, practical researches [91, 94] show that the performance of the system can be managed if not all API and Native API calls are intercepted. In other words, problems associated with the system

performance can be avoided by intercepting only the wanted calls at both levels. Therefore, Deviare API and the Kernel rootkit should be used to intercept only the selected calls in order to avoid the performance problem.

5.6 Summary

In summary, the implementation of this approach that requires information from two different levels has been explained in this chapter, namely: the user and kernel level. Tempura, known as the main tool of this approach, has been used to combine the information coming from both levels. However, when only one of the levels is used, whether it is the user or the kernel level, it has its own drawbacks and provides incomplete knowledge to Tempura. Consequently, merging the two levels by Tempura will close the gap.

Each level has been explained separately and a full explanation of its functions and tools, inputs, etc. has been shown. Firstly, at the user level a tool called Nektra's Deviare API has been used to serve the needs of Tempura. This tool provides each API call with its parameters, process ID, and the file which issues the call. Tempura will receive the information from this tool and examine it. Only wanted API calls will be intercepted by Deviare API and only interested parameters will be examined by Tempura.

At the kernel level, Deviare API will play the role of the kernel rootkit and has been used to provide some information in case a direct call has been issued to the

kernel. The role of the kernel rootkit is to intercept some selected Native API calls with their inputs and deliver them to Tempura. On the other hand, it has been shown that some researches claim that choosing just one level to trace system calls to detect computer viruses might lead to the release of some of these viruses making them bypass the system. Thus, Tempura will join the information coming from both sides to get a better result making sure that there is no direct call to the kernel without knowledge of this detection system.

The performance of the system has been taken into account because previous research has shown that overload in the system may occur when both levels have been used to trace API and Native API calls. This problem has been addressed in this chapter and it has shown by other researches that the problem can be solved by not intercepting all the calls at both levels. However, research analysis and experiments that used to evaluate of this research, including the performance of the system, will be detailed in the next chapter.

Chapter 6

Case Studies

Objectives:

- Highlight the experiments we conducted to test the prototype implementation presented in Chapter 5.
- Discuss the theory validation of this research and how it can be met.
- Provide the samples used in this research to analyse normal and viral processes.

6.1 Introduction

A number of investigations have been carried out to examine the theory of this research. The comprehensive attempt of a number of computer viruses to attach themselves to other files has been analysed, as presented in Chapter 4. It has been shown that a virus, in order to attach itself to another file will go through five distinct steps. Moreover, some steps have their own rules that need to be satisfied in order to complete the whole attempt of attachment.

A number of 339 virus and normal samples have been examined in this research. All the virus samples have been scanned through Kaspersky Antivirus software [96] to confirm their names and authenticity. In addition, the usability of the system while these samples are being tested will also be considered. Therefore, three different factors will be examined in this experiment namely, false positives, false negatives and system performance and usability.

The test for computer viruses was carried out on a virtual machine software running Microsoft Windows XP. The test for normal processes was carried out on two separate desktops, one of them running Microsoft Windows XP and the other running Windows 7. The reason for running these two different Windows Operating Systems was to examine the reliability of our prototype on both operating systems. In addition, the performance and usability of the system was tested on both operating systems to examine how they perform while the samples are tested.

6.2 Validation of Research Theory

To validate the theory of this research the five steps, which were described in the previous chapter and which represent the whole attachment of a computer virus to another file, need to be tested through different virus classification. More significantly, it is essential to show that these five steps with their rules do not occur with normal processes. However, these two tests will indicate whether or not the five categories can be used to distinguish between normal processes and computer viruses.

Several viruses, commonly used applications and operating system processes were tested to validate the theory of this research. Both Native and win32 API calls with their parameters were analysed and recorded as explained previously. As shown in Figure 4.2, several tools were used to extract these system calls. As a result, the five steps that a virus carries out to attach itself to a new or existing file will be examined in both viruses and benign processes.

6.2.1 Normal Processes Experiment

A collection of 50 normal process samples have been tested in this theory validation to show that these normal processes have not carried out the same steps of virus behaviour which lead to false positives. These processes are listed below in Table 6.1, and they were chosen because they are the active processes on Windows operating systems.

Two desktops with two different Microsoft Windows's operating systems have been used to examine these normal processes. Both computers have full access to the

Internet with the popular up-to-date desktop and Internet applications. The reason for using all these applications when testing the normal processes is also to test the usability as a real time monitor and detector. In addition, Antivirus software was running on both desktops during testing in order to test the usability of the system if both detectors are running.

The processes that are active when the computer is running have been chosen for testing. In addition, the processes that are known by practice and by previous research [45] to issue API calls similar to the ones that are issued by computer viruses have also been chosen for testing such as, svchost.exe. When testing, more than one process can be analysed by this approach's prototype at the same time.

Table 6.1: Normal Processes.

systeminfo.exe	systray.exe
alg.exe	clipsrv.exe
services.exe	msdtc.exe
imapi.exe	lsass.exe
cisvc.exe	svchost.exe
dmadmin.exe	mqsvc.exe
mqtgsvc.exe	dllhost.exe
netdde.exe	igfxsrvc.exe
firefox.exe	smlogsvc.exe
googledrivesync.exe	sessmgr.exe
locator.exe	regsvc.exe
scardsvr.exe	snmp.exe
atsvc.exe	mstask.exe
tapisrv.exe	explorer.exe
taskmgr.exe	hkcmd.exe
msiexec.exe	wmiapsrv.exe
taskman.exe	wuauclt.exe
ctfmon.exe	spoolsv.exe
winlogon.exe	csrss.exe
smss.exe	iexplore.exe
winmsd.exe	msseces.exe
chrome.exe	tracert.exe
tasklist.exe	sort.exe
rsvp.exe	ping.exe

Cipher.exe	compact.exe

6.2.2 Computer Viruses Experiment

A collection of 289 virus samples has been collected from different malware repositories [86, 87]. The collection of viruses has been chosen from various categories of computer viruses, namely: Windows 32 Viruses, Peer to Peer Worms, Network Worms, Email Worms and Instant Messaging Worms. Therefore, as long as the malware attach itself to another file and follows the steps of behaviour which have been previously explained then the aim is that it would be detected by our approach.

There were two experiments carried out for computer viruses, namely, virus analysis which was explained earlier in Chapter 4 and virus detection, the results of which will be explained in more detail in the next chapter. The former experiment was carried out using existing tools to specify the steps of behaviour of computer viruses (Figure 4.2). As shown in Table 6.2 and 6.3, these are the 283 virus samples which were analysed to validate the theory of attachment, i.e., to examine if these viruses have attached themselves to other files.

Table 6.2: Test Viruses for Theory Validation-1.

Malware name	**Malware name**
Malware.Win32/Alcra	Malware.Win32/Bropia
Malware.Win32/Allaple	Malware.Win32/BugBear
Malware.Win32/Areses	Malware.Win32/Chir
Malware.Win32/Bagle	Malware.Win32/Cabanas
Malware.Win32/Evaman	Malware.Win32/Mugly
Malware.Win32/Gibe	Malware.Win32/Poebot
Malware.Win32/Tutiam	Malware.Win32/VB.CA
Malware.Win32/Rbot	Malware.Win32/Mytob
Malware.Win32/Rontokbro	Malware.Win32/Sober
Malware.Win32/Locksky	Malware.Win32/Elkern
Malware.Win32/Korgo	Malware.Win32/Flopcopy
Malware.Win32/Zafi	Malware.Win32/Vanbot

Malware.Win32/Vote	Malware.Win32/Stration
Malware.Win32/Sobig	Malware.Win32/Sohanad
Malware.Win32/Ska	Malware.Win32/Skudex
Malware.Win32/Redesi	Malware.Win32/Paukor
Malware.Win32/Neveg	Malware.Win32/Netsky
Malware.Win32/Myparty	Malware.Win32/Moonlight
Malware.Win32/Minusi	Malware.Win32/Mapson
Malware.Win32/Lovgate	Malware.Win32/Looked
Malware.Win32/Lioten	Malware.Win32/Holar
Malware.Win32/Hocgaly	Malware.Win32/Higuy
Malware.Win32/Gnuman	Malware.Win32/Frethem
Malware.Win32/Fix2100	Malware.Win32/ExploreZip
Malware.Win32/Gain	Malware.Win32/Lirva
Malware.Win32/Blaster	Malware.Win32/Hybris
Malware.Win32/Antiman	Malware.Win32/Cloner
Malware.Win32/Cervivec	Malware.Win32/Tenga
Malware.Win32/Badtrans	Malware.Win32/Doomjuice
Malware.Win32/Hai	Malware.Win32/Funner
Malware.Win32/Reatle	Malware.Win32/Rinbot
Malware.Win32/Polip	Malware.Win32/Puce
Malware.Win32/Parite	Malware.Win32/Qaz
Malware.Win32/Nachi	Malware.Win32/Mywife
Malware.Win32/Magistr	Malware.Win32/Maslan
Malware.Win32/Klez	Malware.Win32/Kipis
Malware.Win32/Kidala	Malware.Win32/Gurong
Malware.Win32/Funlove	Malware.Win32/Padobot
Malware.Win32/Bozori	Malware.Win32/Bofra
Malware.Win32/Sality	Malware.Win32/Anzae
Malware.Win32/Golten	Malware.Win32/Myfip
Malware.Win32/Philis	Malware.Win32/Theals
Malware.Win32/Tirbot	Malware.Win32/Savage
Malware.Win32/Autex	Malware.Win32/Backterra
Malware.Win32/Deborm	Malware.Win32/Dedler
Malware.Win32/Cone	Malware.Win32/Chimo
Malware.Win32/Fanbot	Malware.Win32/Floppy
Malware.Win32/Heretic	Malware.Win32/Hotlix
Malware.Win32/Zusha	Malware.Win32/Jupir
Malware.Win32/Kelvir	Malware.Win32/Kindal
Malware.Win32/MTX-m	Malware.Win32/MyLife
Malware.Win32/Snapper	Malware.Win32/Zindos
Malware.Win32/Annil	Malware.Win32/Antiqfx
Malware.Win32/Onamu	Malware.Win32/PrettyPark
Malware.Win32/Xddtray	Malware.Win32/Maddis
Malware.Win32/Apsiv	Malware.Win32/Benjamin
Malware.Win32/Choke	Malware.Win32/Dabber
Malware.Win32/Dipnet	Malware.Win32/Donk
Malware.Win32/Gregcenter	Malware.Win32/Imbiat
Malware.Win32/HLLP.DeTroie	Malware.Win32/Kelino

Malware.Win32/Logpole	Malware.Win32/Loxar
Malware.Win32/Mellon	Malware.Win32/Misodene
Malware.Win32/Neklace	Malware.Win32/Pepex
Malware.Win32/RAHack	Malware.Win32/Randin
Malware.Win32/Rirc	Malware.Win32/Stator
Malware.Win32/Tumbi	Malware.Win32/Datom
Malware.Win32/Tzet	Malware.Win32/Zar
Malware.Win32/Unfunner	Malware.Win32/Yanz
Malware.Win32/Upering	Malware.Win32/Wozer
Malware.Win32/Vavico	Malware.Win32/Warpigs
Malware.Win32/Visilin	Malware.Win32/Wallz

Table 6.3: Test Viruses for Theory Validation-2.

Malware name	**Malware name**
Malware.Win32/Codbot	Malware.Win32/Eliles
Malware.Win32/Detnat	Malware.Win32/Eyeveg
Malware.Win32/Darby	Malware.Win32/Feebs
Malware.Win32/Dumaru	Malware.Win32/Forbot
Malware.Win32/Fizzer	Malware.Win32/Fujacks
Malware.Win32/Mydoom	Malware.Win32/Wukill
Malware.Win32/Spybot	Malware.Win32/Sdbot
Malware.Win32/Oddbob	Malware.Win32/Agobot
Malware.Win32/Mocbot	Malware.Win32/Mimail
Malware.Win32/Bobax	Malware.Win32/Zotob
Malware.Win32/Aimbot	Malware.Win32/Yaha
Malware.Win32/Wootbot	Malware.Win32/Virut
Malware.Win32/Pesin	Malware.Win32/Small
Malware.Win32/Sircam	Malware.Win32/Sixem
Malware.Win32/Ritdoor	Malware.Win32/Reper
Malware.Win32/Oror	Malware.Win32/Outa
Malware.Win32/Mytobor	Malware.Win32/Mypics
Malware.Win32/Monikey	Malware.Win32/Mobler
Malware.Win32/Maldal	Malware.Win32/Melissa
Malware.Win32/Lolol	Malware.Win32/Satir
Malware.Win32/Gokar	Malware.Win32/Hantaner
Malware.Win32/Goner	Malware.Win32/Heidi
Malware.Win32/Fbound	Malware.Win32/Banwarum
Malware.Win32/Beast	Malware.Win32/Antiax
Malware.Win32/Blebla	Malware.Win32/Apost
Malware.Win32/Ircbot	Malware.Win32/Appflet
Malware.Win32/Torvil	Malware.Win32/Traxg
Malware.Win32/Valla	Malware.Win32/Qeds
Malware.Win32/Womble	Malware.Win32/Serflog
Malware.Win32/Flukan	Malware.Win32/Fatcat
Malware.Win32/Sasser	Malware.Win32/Plexus
Malware.Win32/Rants	Malware.Win32/Opaserv
Malware.Win32/Nugache	Malware.Win32/Nimda

Malware.Win32/Mabutu	Malware.Win32/Luder
Malware.Win32/Lovelorn	Malware.Win32/Kriz
Malware.Win32/Jeefo	Malware.Win32/Kebede
Malware.Win32/Inforyou	Malware.Win32/Ganda
Malware.Win32/Nanspy	Malware.Win32/Cissi
Malware.Win32/Bagz	Malware.Win32/Atak
Malware.Win32/Delf	Malware.Win32/Braid
Malware.Win32/Opanki	Malware.Win32/Aliz
Malware.Win32/Tenrobot	Malware.Win32/Swen
Malware.Win32/Pinom	Malware.Win32/Assasin
Malware.Win32/Deloder	Malware.Win32/Capside
Malware.Win32/Derdero	Malware.Win32/Doep
Malware.Win32/Bube	Malware.Win32/Drefir
Malware.Win32/Guap	Malware.Win32/Harwig
Malware.Win32/HPS	Malware.Win32/Tibick
Malware.Win32/Kalel	Malware.Win32/Kassbot
Malware.Win32/Aplore	Malware.Win32/Licu
Malware.Win32/Raleka	Malware.Win32/Randex
Malware.Win32/Ahker	Malware.Win32/Anap
Malware.Win32/Cuebot	Malware.Win32/Deadcode
Malware.Win32/Primat	Malware.Win32/Protoride
Malware.Win32/Mofei	Malware.Win32/Antinny
Malware.Win32/Bereb	Malware.Win32/Bilay
Malware.Win32/Darker	Malware.Win32/Buchon
Malware.Win32/Faisal	Malware.Win32/Francette
Malware.Win32/Jared	Malware.Win32/ Jitux
Malware.Win32/Krepper	Malware.Win32/Lacrow
Malware.Win32/LyndEgg	Malware.Win32/Magold
Malware.Win32/Mona	Malware.Win32/Navidad
Malware.Win32/PMX	Malware.Win32/Qizy
Malware.Win32/Reur	Malware.Win32/Salga
Malware.Win32/Tanked	Malware.Win32/Titog
Malware.Win32/Allocup	Malware.Win32/Amus
Malware.Win32/Envid	Malware.Win32/Shuck
Malware.Win32/Looksky	Malware.Win32/Smibag
Malware.Win32/Semapi	Malware.Win32/Smeagol
Malware.Win32/Seppuku	Malware.Win32/Silva
Malware.Win32/Shodabot	

The other experiment was to show that these steps of virus behaviour can be used to detect computer viruses by using the prototype of this research as explained in Chapter 5. A number of chosen viruses from those analysed before have been observed and tested by the prototype. In addition, a collection of viruses which are

unknown to the prototype has also been examined. The unknown ones have been chosen to test whether the prototype is able to detect previously unseen viruses which is one of the objectives of this research.

Despite the fact that the prototype of this research can analyse more than one process at the same time, the computer viruses have been tested separately. Therefore, the viruses have been tested one by one and this has been done by restoring the virtual machine each time a virus is tested. The reason for that is to ensure a clean operating system for each virus and not make them interfere with each other. The collection of the tested viruses is shown below in Table 6.4.

Table 6.4: Test Viruses for Detection Experiment.

Malware name	**Malware name**
Malware.Win32/Klez	Malware.Win32/Watcher
Malware.Win32/Zori	Malware.Win32/Rega
Malware.Win32/Borzella	Malware.Win32/Weakas
Malware.Win32/Eliles	Malware.Win32/Eyeveg
Malware.Win32/Feebs	Malware.Win32/Qizy

6.2.3 System Performance and Usability

In this experiment, two criteria have been tested which are system performance and usability. Under normal computer use, two actual computer desktops have been observed with the prototype of this research installed on them to carry out these two experiments. The two computers have full access to the Internet and have the normal popular desktop and Internet applications. There is also Antivirus software installed on these two desktops when testing the system performance and usability to observe the ability of this research's prototype to work alongside other virus detections.

With the system performance, the speed of the operating system is observed while the prototype is running. In spite of the fact that there are several tools to observe the performance of the system, Microsoft Windows Operating systems have the well-known application, Windows Task Manager, which gives us the ability to monitor the system performance [97], as shown subsequently in Figure 6.1 and 6.2. Windows Task Manager can measure the CPU and Physical Memory (MB) usage.

In addition, Resource Monitor is a part of Windows Task Manager that monitors the resources of the system at the same time, such as CPU, memory, disk and network. As a result, the performance of the system will be observed using this application during the runtime monitoring of this prototype. This experiment is done to show how the system operates during the runtime monitoring.

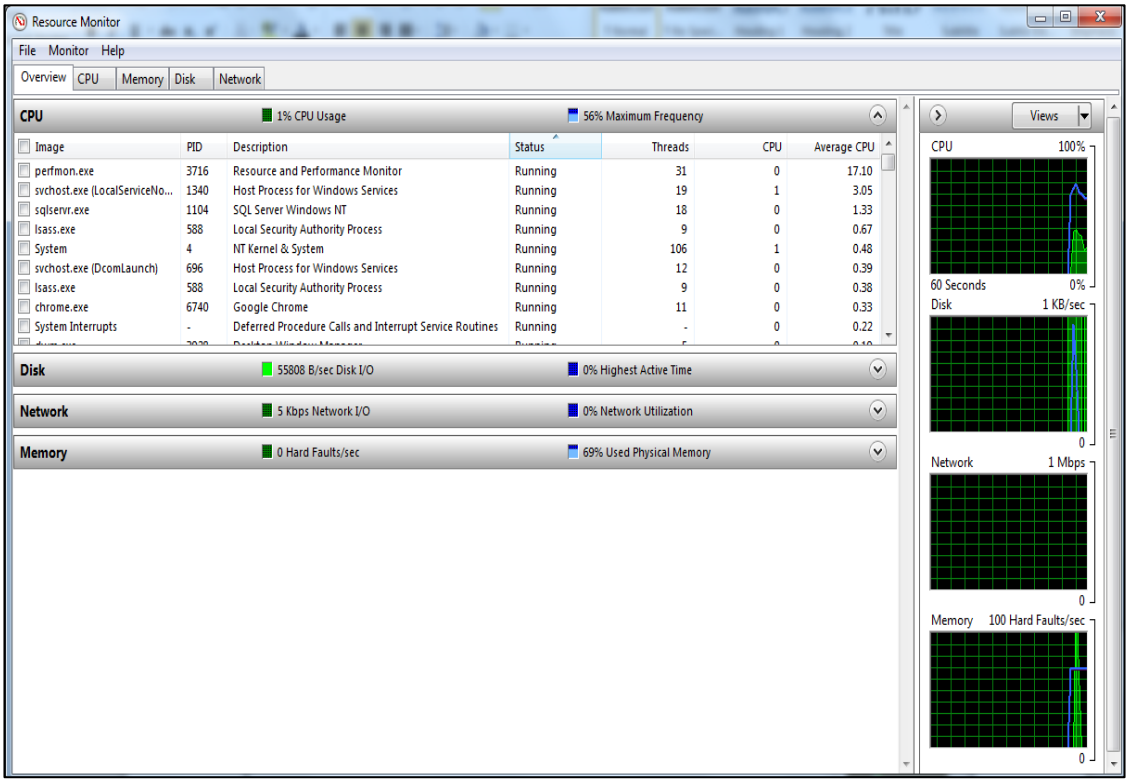


Figure 6.1: Resource Monitor.

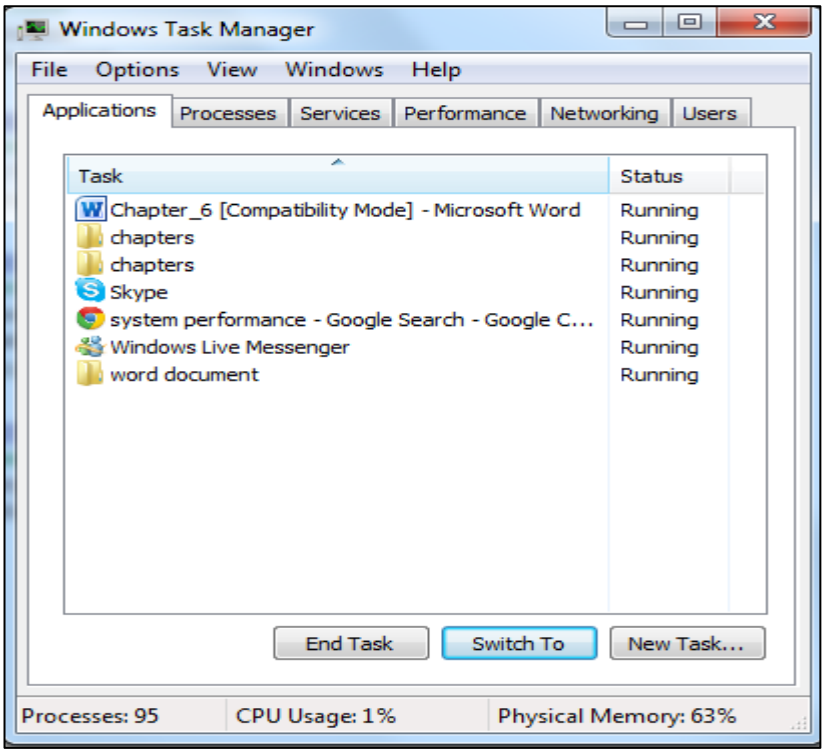


Figure 6.2: Windows Task Manager.

The other experiment is the usability of both computer desktops during the runtime monitoring. What is meant by usability here is using the applications of both operating systems at the same time as monitoring. In other words, can the applications of the operating system be used while the runtime monitoring is running?

Several desktop applications running on two different Windows operating systems (Windows XP and 7), have been observed in this experiment such as, Microsoft Word and Microsoft Visio etc. In addition, the usability of applications which work with the Internet is also observed. Therefore, both Internet and desktop applications have been monitored to test their usability when the prototype of this research observes the system processes. Accessing and using Internet and desktop applications is significant to the end user because it will not be sufficient to protect the

operating system from computer viruses and freezing at the same time, resulting in the user not being able to access the applications during the monitoring.

Therefore, both system performance and usability experiments were carried out to test and evaluate the speed response of the system and its ability to use both desktop and Internet applications. However, the results of the two experiments will be significant in this research due to the fact that the system should be accessible and useable by the end user of the OS.

6.3 Summary

In this chapter, several experiments have been detailed to show how theory validation of this research can be met. Three main experiments were carried out in this research, namely: virus detection, normal processes and system performance and usability. Each of them were itemised to show how the evaluation and results of this research can be accomplished. A total of 339 normal and virus samples were chosen to test the prototype of this research.

The first experiment was the normal processes experiment. In this experiment a collection of 50 normal processes samples were tested to show that the previously observed five steps to detect computer viruses will not provide a false positive and detect normal processes. Two computer desktops with two different operating systems were installed to test these normal processes with the prototype of this research. The most executed processes within the two operating systems were chosen as well as those similar to computer viruses.

The second experiment was virus analysis and detection. In the virus analysis experiment a collection of 283 virus samples were analysed to observe their Win32 and Native API calls and provide the steps of behaviour that computer viruses do. The virus analysis is done by using several tools as explained in Chapter 4. The virus detection experiment was done by using the prototype of this research to examine selected virus samples from the analysed one. In addition, previously unseen viruses by the prototype were also tested to show its ability to detect them.

The last experiment was to test both system performance and usability. Two computer desktops which have full access to the Internet and have the normal popular desktop and Internet applications were used in both experiments. The system performance was done to test the speed of the operating system with the CPU and physical memory usage while the runtime monitoring is running. The well-known Microsoft Application Windows Task Manager is used to test the performance of the system. The other experiment was the usability of the Internet and desktop applications while the prototype is running.

The results of these experiments will be explained in detail in the next chapter. The next chapter will discuss the research results and evaluation. It will show the outcomes of this research including normal and virus samples.

Chapter 7

Results: Analysis and Evaluation

Objectives:

- Discuss the results of both virus analysis and the prototype.
- Evaluate the research that has been explored in this thesis.
- Provide a discussion of the dataset selection and virus detection.
- Discuss the limitations of the proposed approach with their solutions.

7.1 Introduction

While Chapters 4 and 5 discussed issues related to the framework and implementation of this research and how the components of the architecture connect with each other, this chapter examines the overall results including virus detection, normal process results and finally the system's performance and usability. It then evaluates this approach based on the research hypotheses and the evaluation criteria:

- Detecting known and unknown viruses.
- False positive production.
- Running with an acceptable system performance.
- The ability for users to interact with the system while the prototype is running.

Carrying out the experiments was a long and arduous process. To guarantee a clean environment for the following test, analysing and testing a computer virus necessitates various re-installations of the host machine. To achieve this, the VirtualBox virtual machine (VM) was restored to a clean state each time the analysis and testing of a virus was completed. The virus free environment for each virus test was required to ensure that the virus would execute normally and not be affected by a previous infection of the virtual machine.

This chapter is organised as follows. The results of the analysis of the computer viruses detailed in Section 4.3.2 are discussed in Section 7.2. Section 7.3 then discusses the results for this research prototype, including normal processes and the results for both known and unknown viruses. Section 7.4 examines whether the system was

running with acceptable performance, as well as the ability to use the system's resources while the prototype is running. Next, an evaluation of this research is made in Section 7.5, followed by a discussion of the implications in Section 7.6. Finally, in Section 7.7, the limitations of the research are discussed and solutions offered.

7.2 Analysis Results

It can be argued that the results of the virus analysis may not be significant and that they need not be included in the overall results. However, without these results, the theoretical aims of this research would not be accomplished. In other words, these results are the foundation on which the theory of attaching a computer virus to another file or files is based. The results of the analysis of 283 viruses using existing tools, as explained in Chapter 4, are shown in Figure 7.1.

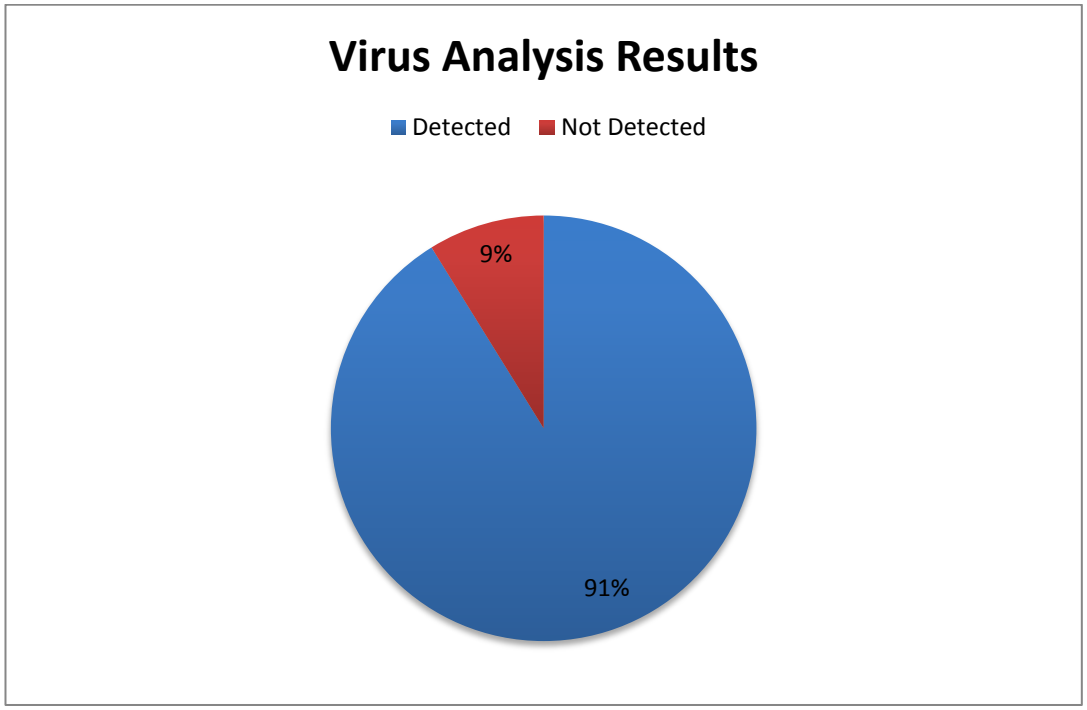


Figure 7.1: Results of virus analysis.

The results listed above indicate that the majority of the viruses in the various classes did indeed attach themselves to other new or old files. However, several viruses contradicted the theory of this research by not doing so. That is to say, the rules governing these malwares did not follow the five steps on which this research is based. Two possible explanations are proposed. The malware may have failed to find a suitable environment or victim file to attach itself to [23]. In other words, a virus needs help to spread, whether it is the system service of a file or some other type of support; if it does not find this assistance, it will not infect the system. Alternatively, the codes may not attach themselves to other files at all and therefore cannot be considered computer viruses according to the definitions adopted here [1, 16]. This means that they can be considered to be some other type of malware but not computer viruses. Such malware would not be targeted or detected by the present approach, as clarified in Chapter 1.

7.3 Prototype Results

This section is divided into two subsections, giving the results of normal processes and of virus detection, which together indicate whether the prototype has the ability to detect viruses without false negatives and whether it can distinguish between viruses and normal processes by not producing false positives.

7.3.1 Normal Processes

As explained earlier, the samples tested were 50 normal processes known to be the active during the runtime monitoring of the prototype. These were examined at

runtime by means of both Deviare API, as explained in Section 5.3.1, and Tempura, as detailed in Section 5.3.4. Both API and Native API calls were extracted at runtime and delivered to Tempura to examine them. With each process, its process ID, its name, the Native or API call issued and parameters associated with some of these calls were examined each time a call was issued.

As mentioned in Chapter 5, the role of Tempura was to examine these processes to determine whether they followed the theory of attachment and issued calls related to the five categories with their rules. The testing of normal processes took approximately five days to complete and the results are shown in Table 7.1. The first and third columns list the names of the normal process samples which were tested, while the second and fourth columns state whether each process attached itself to another file or file. “Yes” would mean that the process did follow the theory by attaching itself to another file or files and was therefore detected by the prototype.

Table 7.1: Test Results for Normal Processes.

Process Name	Detected	Process Name	Detected
systeminfo.exe	No	systray.exe	No
alg.exe	No	clipsrv.exe	No
services.exe	No	msdtc.exe	No
imapi.exe	No	lsass.exe	No
cisvc.exe	No	svchost.exe	No
dmadmin.exe	No	mqsvc.exe	No
mqtgsvc.exe	No	dllhost.exe	No
netdde.exe	No	igfxsrvc.exe	No
firefox.exe	No	smlogsvc.exe	No
googledrivesync.exe	No	sessmgr.exe	No
locator.exe	No	regsvc.exe	No
scardsvr.exe	No	snmp.exe	No
atsvc.exe	No	mstask.exe	No
tapisrv.exe	No	explorer.exe	No
taskmgr.exe	No	hkcmd.exe	No
msiexec.exe	No	wmiapsrv.exe	No
taskman.exe	No	wuauclt.exe	No

ctfmon.exe	No	spoolsv.exe	No
winlogon.exe	No	csrss.exe	No
smss.exe	No	iexplore.exe	No
winmsd.exe	No	msseces.exe	No
chrome.exe	No	tracert.exe	No
tasklist.exe	No	sort.exe	No
rsvp.exe	No	ping.exe	No
cipher.exe	No	compact.exe	No

In fact, the table shows that none of these normal processes attempted to attach itself to another file and that none of the issued calls related to all of the five categories with their rules. While some of these normal processes did issue calls from the five categories, none did so from all of the categories and in the order that a computer virus would follow, despite attempts to make the processes demonstrate various forms of behaviour by simulating as many typical user interactions with each normal process as possible. However, due to the fact that these normal processes have access to the Internet, they can be infected from outside. In other words, some viruses have the ability to detect running processes and then use them to serve the virus needs such as terminate or erase it [23]. Hence, if a virus detects such a process it will behave as a benign process and not like virus behaviour, including attachment to other files. These viruses will not be detected by our prototype because they do not act like viruses.

Listing 7.1, for example, shows that the svchost.exe process issued Native and API calls from certain categories but that it did not do so in any way closely following the theory of this research, which would have made it appear to be a computer virus. The fact that the test was carried out on all of the normal processes and that none

was found to attach itself to another file means that Tempura reported no false positives.

Listing 7.1: Snapshot of svchost.exe log file.

```
State  18: Call="NtWriteFile"                    *
State  18: S=0
State  18: ProcessName="svchost.exe"
State  20: Call="NtClose"
State  20: S=0
State  20: ProcessName="svchost.exe"
State  22: Call="GetFileType"                    *
State  22: S=0
State  22: ProcessName="svchost.exe"
State  22:  PID:1356 issued a call in cat2
State  24: Call="NtReadFile"
State  24: S=7
State  24: ProcessName="svchost.exe"
State  26: Call="NtReadFile"
State  26: S=0
State  26: ProcessName="svchost.exe"
```

The snapshot above for the svchost.exe log file indicates that this normal process issued calls belonging to at different categories. For example, in state 18, it issued a Native API call "NtWriteFile" belonging to category four (Write and/or

Delete), but this is not significant, because in previous states it did not issue calls from categories one to three. However, in state 22, the process issued an API call "GetFileType" belonging to category two, which looked promising and might concern Tempura. In this case, Tempura will have waited to for further calls to see whether the five categories were completed. It can be seen that the process did not subsequently issue calls in the order normally followed by a virus when attaching itself to other files, so Tempura will have returned to the beginning to search for calls from the five categories in the order typical of a virus.

Given that none of these processes attempted to attach itself to another file, the five categories derived from an analysis of viruses can be used to distinguish between normal and viral processes. In addition, the rules associated with each category have proved to be significant in differentiating benign from viral processes. For example, it is apparent from Listing 7.1 that processes sometimes issue Native and API calls from different categories, but that they do not follow the order which computer viruses do to spread through the operating system. In addition, while processes may sometimes follow this order, they appear not to read themselves, which viruses do, as explained in Section 4.3.2.

To clarify the idea, Listing 7.2 shows a snapshot from another process, services.exe, which issued Native API calls from categories 2, 1 and 3 in states 760, 766 and 770 respectively. The order of these calls exactly resembles those that would be made by a virus, but when the open file was received, Tempura examined the parameters and found that the opened file was not the same as the process which

issued the call. Therefore, Tempura decided that this process was not viral and started searching again, as shown in state 772.

Listing 7.2: Snapshot of services.exe log file.

```
State 760: Call="GetLongPathNameW"                    *
State 760: S=0
State 760: ProcessName="services.exe"
State 760:  PID:928 issued a call in cat2
State 760: Call="GetLongPathNameW"
State 760: S=1
State 760: ProcessName="services.exe"
State 766: Call="FindFirstFileW"                      *
State 766: S=7
State 766: ProcessName="services.exe"
State 766:  PID:928 issued a call in cat1
State 766: Call="FindFirstFileW"
State 766: S=1
State 766: ProcessName="services.exe"
State 770: Call="NtOpenFile"
State 770: S=8
State 770: ProcessName="services.exe"
State 770: Call="NtOpenFile"
State 770: S=1
State 770: ProcessName="services.exe"
```

```
State 772: Call="NtQueryDirectoryFile"                    *

State 772: S=0

State 772: ProcessName="services.exe"

State 772: Call="NtQueryDirectoryFile"

State 772: S=1

State 772: ProcessName="services.exe"
```

The test results show that none of the 50 normal process samples followed the five steps with the rules associated with these categories. Therefore, it can be said that the theory implemented in this research by Tempura has successfully distinguished between viral and non-viral processes to the extent that no false positives were returned. The following subsection assesses whether or not the prototype successfully detected both known and unknown viruses.

7.3.2 Computer Viruses

As mentioned earlier, each virus was tested separately by the prototype in order to ensure that the results were as clear as possible. In addition, before each virus was examined, the virtual machine was reset to a virus free state that had no virus except the one to be tested, to guarantee that there was no superimposition of one virus on another. Table 7.2 shows the viruses which were tested by the prototype to examine whether Tempura had the ability to detect an attempt at attachment. The known viruses were chosen from a list of 283 viruses, while the anonymous ones were selected from the virus repositories described in Chapter 6.

Table 7.2: Prototype Virus Test Results.

Malware name	Detected	Malware name	Detected
Malware.Win32/Klez	Yes	Malware.Win32/Watcher	Yes
Malware.Win32/Zori	Yes	Malware.Win32/Rega	Yes
Malware.Win32/Borzella	No	Malware.Win32/Weakas	Yes
Malware.Win32/Eliles	Yes	Malware.Win32/Eyeveg	Yes
Malware.Win32/Feebs	No	Malware.Win32/Qizy	Yes

Table 7.2 shows that several viruses attempted to attach themselves to new or existing files and were thus recognised by the prototype. It also shows that the prototype was able to detect both known and unknown computer viruses. Two viruses do appear to have escaped or bypassed the prototype, resulting in false negatives. It is difficult to be certain that these results really were false negatives, however, for the following reasons. First, an undetected virus might not find the proper file that it needed to attach itself to. This justification can be accepted for both known and unknown viruses. A second explanation is that the malware in question might not be considered a virus at all, because it was not intended to attach itself to other files and was therefore of a type which the prototype was not designed to detect. This justification can be accepted only in the case of viruses unknown to the prototype. The final factor to consider here is that a computer virus can behave unexpectedly. For example, it may define its own API calls; that is to say, it may be able to construct its own modified kernel API driver, or it may detect that it runs in a VM and remains "idle". Again, the prototype was not built to deal with such viruses.

The following snapshots, in Listings 7.3 and 7.4, describe how the "Rega.exe" virus can be detected by the prototype. The first snapshot shows how Tempura will read the Native and API calls issued by this virus from Deviare API, while the second

shows how the final result can be achieved when the virus attempts to attach itself to another file and follows the five steps with their rules as described in Chapter 4.

Listing 7.3: Snapshot of Rega.exe Assertion.

```
!PROG: assert Name:Rega.exe:Id:3512:Call:GetFileAttributesW:Inputs::!
!PROG: assert Name:Rega.exe:Id:3512:Call:NtQueryAttributesFile:Inputs::!
!PROG: assert Name:Rega.exe:Id:3512:Call:NtQueryAttributesFile:Inputs::!
!PROG: assert Name:Rega.exe:Id:3512:Call:NtQueryAttributesFile:Inputs::!
!PROG: assert Name:Rega.exe:Id:3512:Call:FindClose:Inputs::!
!PROG: assert Name:Rega.exe:Id:3512:Call:FindFirstFileNameW:Inputs::!
!PROG: assert Name:Rega.exe:Id:3512:Call:NtQueryDirectoryFile:Inputs::!
!PROG: assert Name:Rega.exe:Id:3512:Call:NtQueryDirectoryFile:Inputs::!
!PROG: assert Name:Rega.exe:Id:3512:Call:NtReadFile:Inputs:Rega.exe:!
!PROG: assert Name:Rega.exe:Id:3512:Call:SetFileAttributesW:Inputs::!
!PROG: assert Name:Rega.exe:Id:3512:Call:NtSetInformationFile:Inputs::!
!PROG: assert Name:Rega.exe:Id:3512:Call:WriteFile:Inputs::!
!PROG: assert Name:Rega.exe:Id:3512:Call:NtWriteFile:Inputs::!
```

In Listing 7.4, it can be seen that “Rega.exe” issued calls from the first three categories as shown in states 77, 81 and 84; therefore this PE file is considered to be a candidate virus. In states 85 and 87 the virus issues calls from the last two categories by setting the file attributes and then writing itself to the new file. As described in Chapter 5, when a file follows the five steps with their rules and the *S* variable reaches

5 or 11, then it can be considered to be a virus. Hence, the prototype identified "Rega.exe" as a virus, because S=11, as shown in state 88.

Listing 7.4: Snapshot of Rega.exe log file (after detection).

```
State  77: i=77
State  77: Call="GetFileAttributesW"
State  77: S=0
State  77: ProcessName="Rega.exe"
State  77:  PID:3512 issued a call in cat2                    *
State  81: i=81
State  81: Call="FindClose"
State  81: S=7
State  81: ProcessName="Rega.exe"
State  81:  PID:3512 issued a call in cat1                    *
State  84: i=84
State  84: Call="NtReadFile"
State  84: S=8
State  84: ProcessName="Rega.exe"
State  84:  The file ( (Rega.exe) ) issued calls in the previous two categories
State  84:  And a call issued in cat3                         *
State  84: _____Then_______(it's a candidate virus)_____________
State  85: i=85
State  85: Call="SetFileAttributesW"
State  85: S=9
```

```
State  85: ProcessName="Rega.exe"
State  85:  A call issued in cat5                                    *
State  87: Call="WriteFile"
State  87: S=12
State  87: ProcessName="Rega.exe"
State  87:  And a call issued in cat4                                *
State  88: i=88
State  88: Call="NtWriteFile"
State  88: S=11
State  88: ProcessName="Rega.exe"
State  88:  The file ( (Rega.exe) ) is a COMPUTER VIRUS       *
```

"Watcher.exe" is another example of how the prototype can detect viruses, as shown in Listing 7.5. Tempura reads the Native and API calls in the same way as "Rega.exe" in Listing 7.3. Here, it can be observed that S=5, which indicates that the PE file has issued calls related to all five steps and satisfies their rules, as shown in states 3, 5, 10, 12 and 13. Thus, the file has followed the theory of attachment and can be considered a virus, as shown in state 14.

Listing 7.5: Snapshot of Watcher.exe log file (after detection).

```
State   3: Call="NtQueryDirectoryFile"
State   3: S=0
State   3: ProcessName="Watcher.exe"
```

```
State   3:  PID:2800 issued a call in cat1                    *

State   5: i=5

State   5: Call="GetBinaryTypeW"

State   5: S=1

State   5: ProcessName="Watcher.exe"

State   5:  PID:2800 issued a call in cat2                    *

State  10: i=10

State  10: Call="NtOpenFile"

State  10: S=2

State  10: ProcessName="Watcher.exe"

State  10:  The file ( (Watcher.exe) ) issued calls in the previous two categories

State  10:  And a call issued in cat3                         *

State  10: _____Then_______(it's a candidate virus)__________

State  12: i=12

State  12: Call="NtSetInformationFile"

State  12: S=3

State  12: ProcessName="Watcher.exe"

State  12:  A call issued in cat5                             *

State  13: i=13

State  13: Call="WriteFileEx"

State  13: S=6

State  13: ProcessName="Watcher.exe"

State  13:  And a call issued in cat4                         *
```

```
State  14: i=14

State  14: Call="NtWriteFile"

State  14: S=5

State  14: ProcessName="Watcher.exe"

State  14:  The file ( (Watcher.exe) ) is a COMPUTER VIRUS       *
```

Other viruses tested by the prototype were also detected in the same way as “Rega.exe” and “Watcher.exe” as shown in Appendix A, while those viruses which remained undetected were not recognised by the prototype because they did not issue calls related to the five categories.

7.4 System Performance and Usability

The performance and usability of the system is assessed by testing the normal processes on two desktops, running the Microsoft Windows XP and 7 operating systems respectively. The two desktops were run as if by a normal user running some everyday programs. Figures 7.2, 7.3 and 7.4 show that the performance of both desktops was normal under varied levels of CPU and memory usage. The very slow operation of the prototype caused a high consumption of system resources and there were some occasions when the computer had to be rebooted. There was an observation about which part of our prototype did cause the performance overhead. It was observed that when several tests are carried out, the system performance gets slower and slower. Therefore, the computer had to be rebooted after a number of tests were completed in order to run with a better performance. However, this is

acceptable, because the Window system can run with a very slow performance under normal conditions.

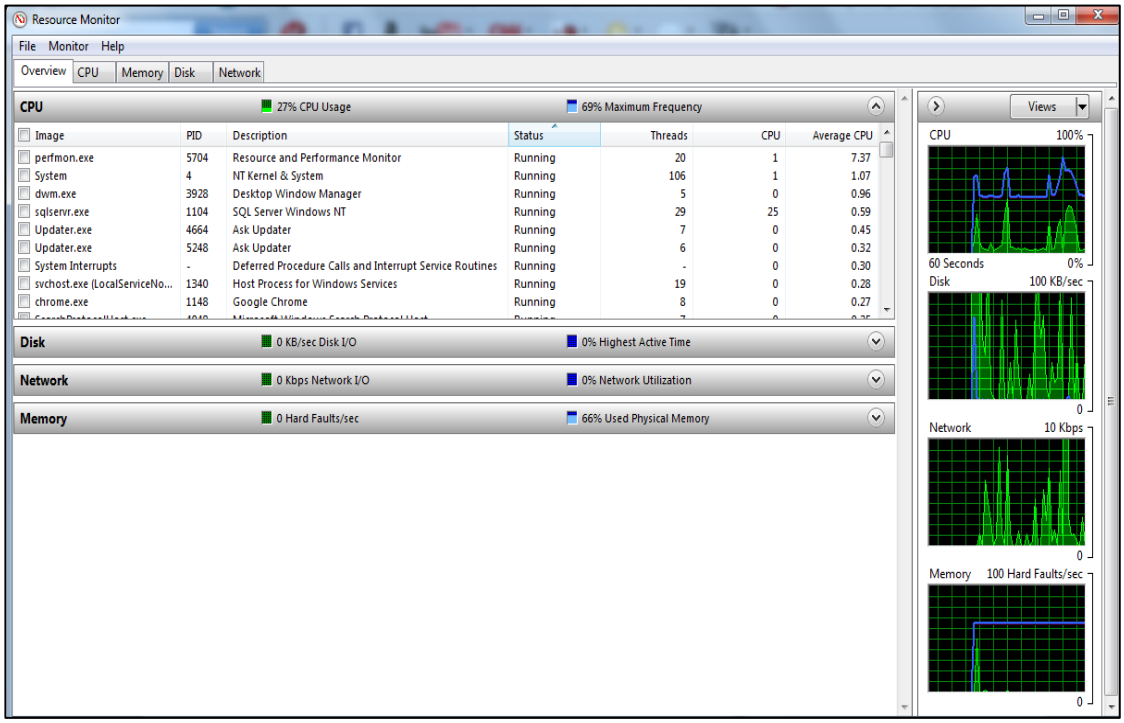


Figure 7.2: Windows 7 Resource Monitor.

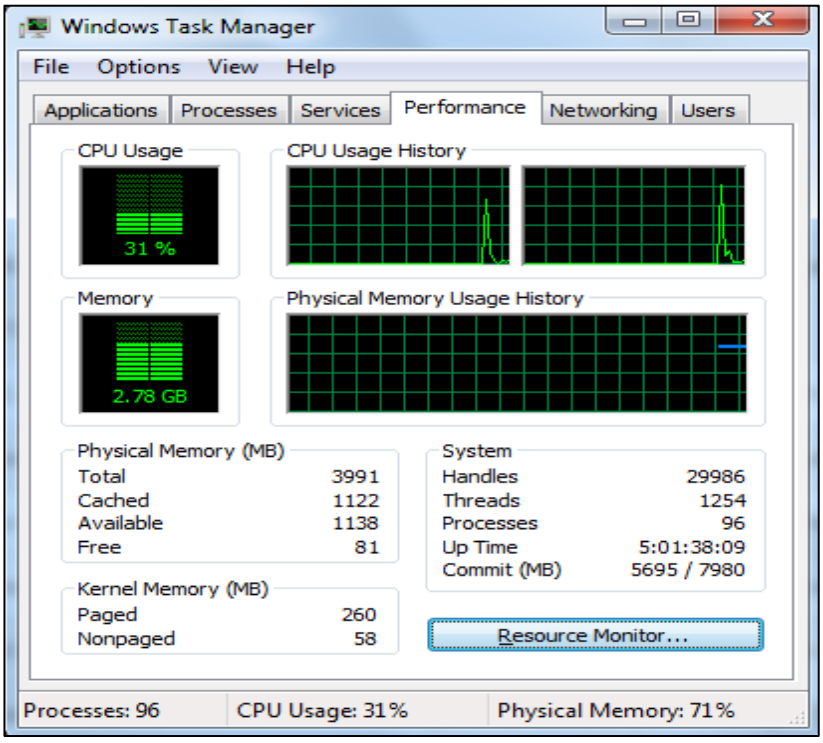


Figure 7.3: Windows 7 Task Manager.

Figure 7.4 shows the performance of the system on Windows XP while the prototype was running. It shows that the CPU usage was 41%, which is acceptable, whereas if it were to reach 100% the operating system would become very busy and freeze. However, the performance of Windows XP was poorer than that of Windows 7, which can be attributed to the fact that the desktop running Windows 7 was a newer model with a better specification; for example, it had a newer processor.

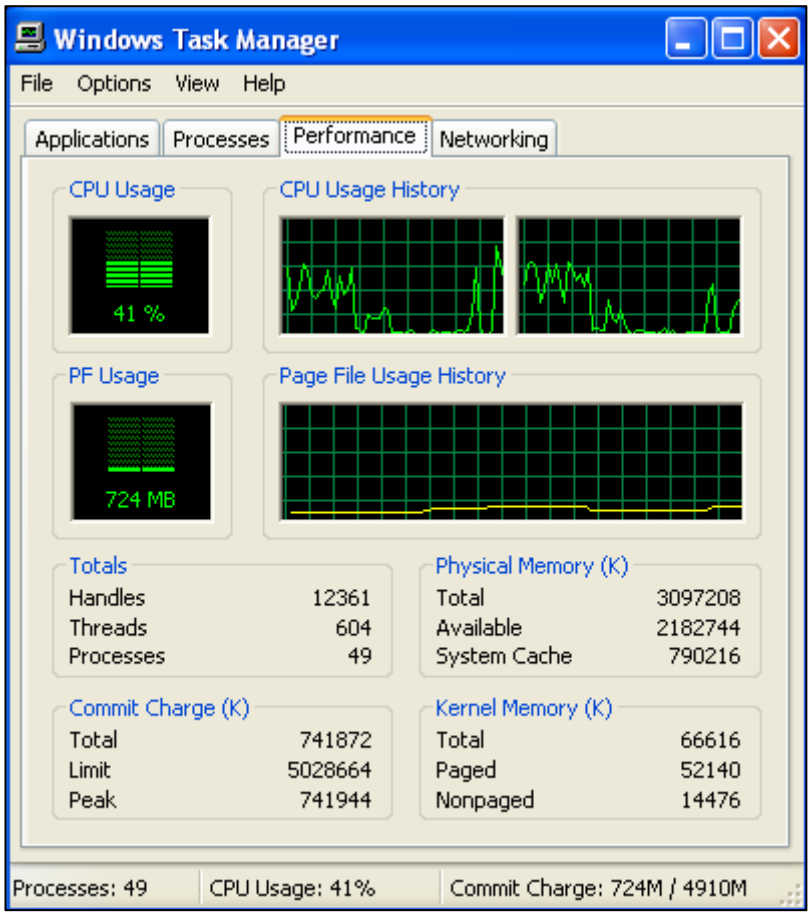


Figure 7.4: Windows XP Task Manager.

The system resources and programs were available to the end-user during the runtime of the prototype. The user was able to use different desktop and Internet applications. A solution to the problem of slow running and the need to reboot will be discussed further at the end of the evaluation section.

7.5 Evaluation

To evaluate the proposed research, a set of experiments were performed on real-world viruses and benign executable processes. These experiments were outlined in Chapter 6 and their results explained in detail in the present chapter. The evaluation of this approach is based on these results with regard to the research hypotheses set out in Chapter 1 and is divided into two parts. The first question is whether or not our prototype consumed a large amount of memory, thus reducing system performance, while the second and more significant one is whether the prototype successfully detected known and unknown viruses.

7.5.1 Performance and Memory Usage

As mentioned in Chapter 1, the number of virus signatures and the ever growing number of computer viruses mean that the storage of signatures required by traditional antivirus software will require increasing memory capacity in the near future. Searching through a signature database for matching viruses is time consuming for the end-user, who is sometimes unable to use the system and internet applications. Therefore, users require light and robust virus detection which uses little system memory, so that the user is able to continue using the system while searching for viruses.

The amount of memory required for our detection prototype, including the installation of all the tools needed, namely Deviare API and its code, AnaTempura, Tempura and the Java pipe, is approximately 6 MB. By comparison, an antivirus

application such as MacAfee [98] requires at least 500 MB free drive space to run on Microsoft Windows, while Kaspersky Antivirus requires at least 352 MB just to store the virus signature database, as noted in Section 1.2. Kaspersky reports that it needs approximately 480 MB free space on the hard drive (depending on the size of the antivirus database). This means that the space needed will grow whenever the program is updated and newly identified signatures need to be stored in the database.

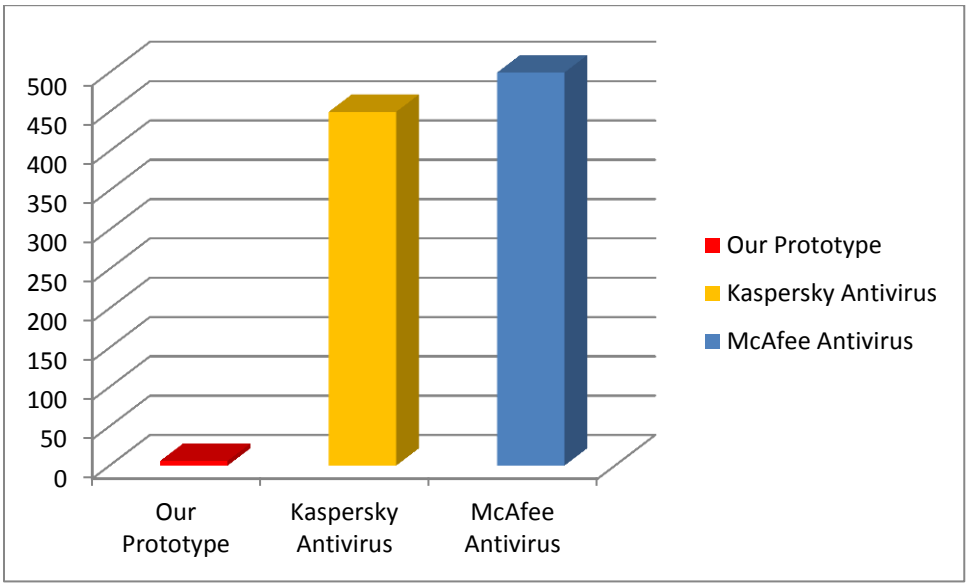


Figure 7.5: Hard drive space needed by our current prototype and other AVs (MB).

Figure 7.5 represents graphically the apparent strong superiority of our prototype over these two commercial antivirus products in terms of hard drive space required. It can be argued, however, that the number of signatures stored in the databases of such products is much greater than the number of viruses tested in this research. In response, a simple calculation can be done to ensure that the comparison is unbiased and valid. According to Yuan [99], there are thousands of API calls in Microsoft Windows. Therefore, it can be assumed that the maximum number of both Win32 and Native API calls will not exceed 100 000. This maximum number should

then be divided by the number of API calls used in our prototype which was 68. The result of this division is approximately 1471, which should then be multiplied by the two kilobytes needed to store the 68 API calls representing the virus behaviours, making approximately 3 MB. When this figure is added to the 6 MB required for installing all the tools with our prototype, it appears that a total of approximately 9 MB free space is needed on the hard drive to store our whole prototype in a Windows operating system, as shown in Figure 7.6.

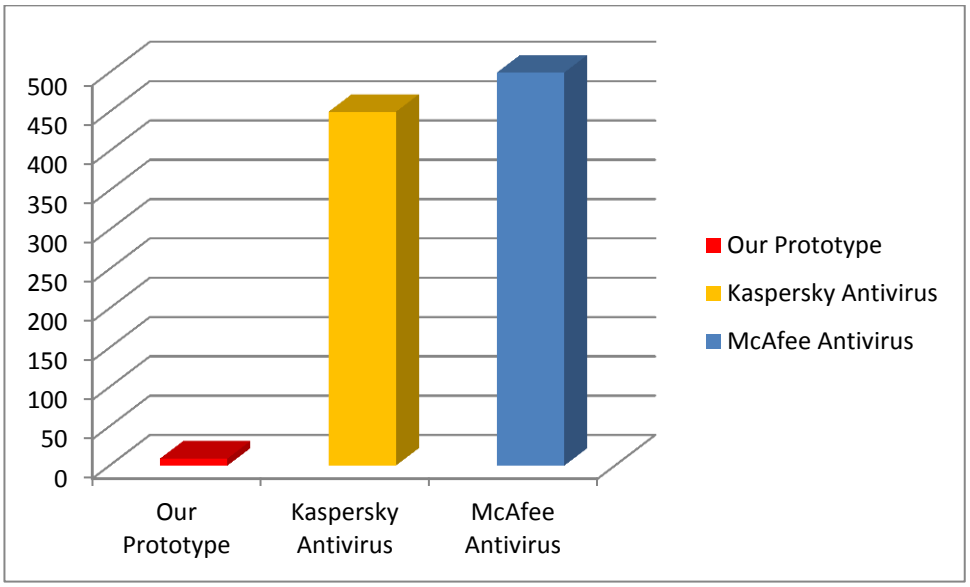


Figure 7.6: Comparison of prototype with other AVs after calculation (MB).

This 9 MB space is needed only if it is assumed that all the API calls will be used as virus behaviours. This and previous research have reported [45] that the functionality of some computer viruses is normally the same. As a result, there will be groups of computer viruses which have the same steps of behaviour and this means that the 9 MB space might be minimised. Consequently, it can be said that our prototype will use less space than traditional antivirus products even if all the API calls are stored in the prototype. The minimum usage of computer memory by our

prototype indicates that the first null hypothesis set out in Section 1.4 is rejected and thus the first alternative hypothesis H_1 is accepted.

H1: Computer viruses can be detected by their behaviours without consuming large hard drive space.

The other factors which are considered significant to the end-user are system performance and usability. An experiment to measure these was conducted on an Intel(R) Core™ i3 CPU M 350 @2.27GHz, running 64-bit Windows 7. To evaluate performance and usability, our prototype was compared with the Microsoft Security Essentials Antivirus software [100].

The full scan took approximately 12 hours to complete, which may be considered a long time, due to poor system performance and usability, which will be detailed in the following sentences. During the full scan of Microsoft Security Essentials Antivirus the system's CPU and memory usage were monitored, as shown in Figure 7.7. In addition, the system's disk and network utilisation were observed using Resource Monitor, as shown in Figure 7.8. The CPU and memory usage varied, but it can be seen that they reached 60% and sometimes 90%. More than one factor affected CPU and memory usage, but the full scan played a major role in keeping the system busy for several reasons, one of which was the large signature database.

Figure 7.8 shows that the network utilisation was normal but that disk activity was at its highest, reaching 100%. Monitoring all these system services demonstrates that during a full scan by an antivirus program, the system may be at its most active

and busiest. It can be said that while several factors influence system activity, a full antivirus scan represents one of the strongest influences.

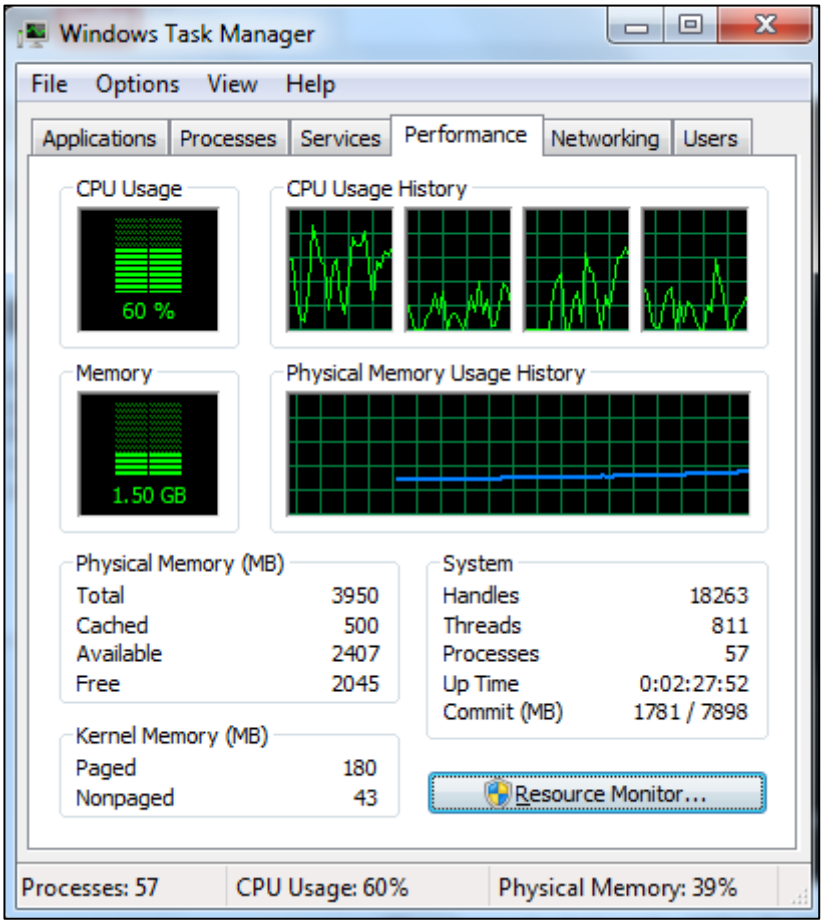


Figure 7.7: Windows Task Manager during Full Scan.

After comparing the Microsoft Security Essentials Antivirus software with our prototype in terms of system performance and usability, as discussed in Section 7.4, it can be said that there are similarities between them and that our prototype has the slight advantage that it can be developed to enhance system performance and usability. The enhancement of our prototype would require a better design of the system, which is believed to be realisable.

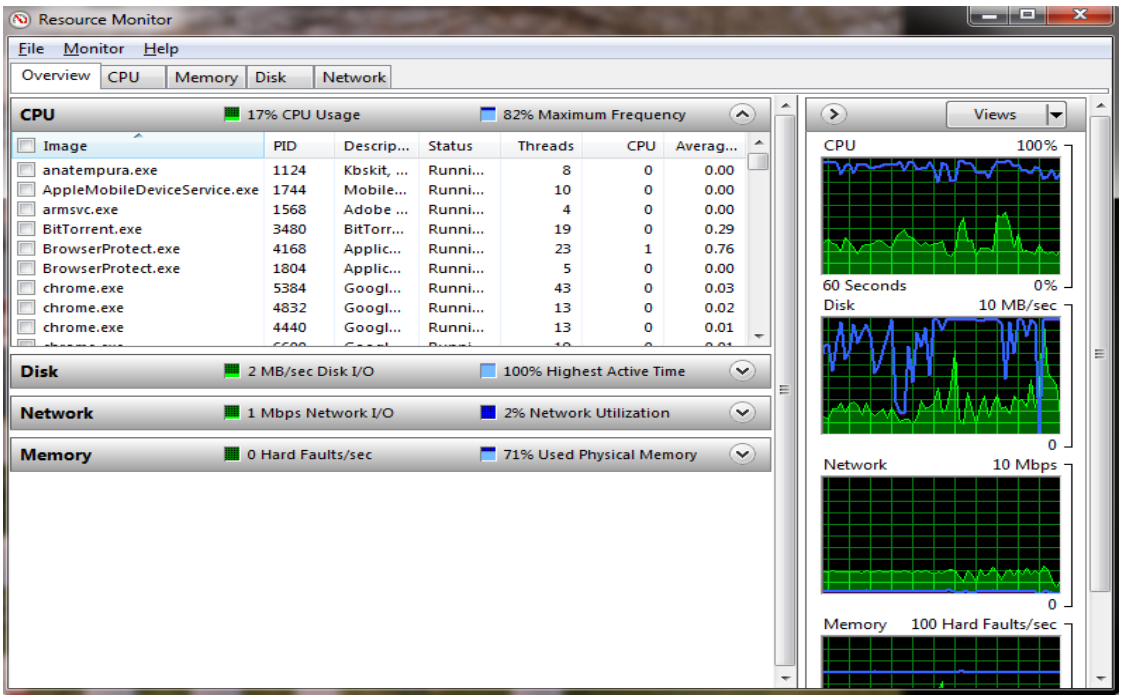


Figure 7.8: Resource monitor during full scan.

7.5.2 Detecting Known and Unknown Viruses

This section discusses the testing of the second null hypothesis explained in Section 1.4, to determine whether or not our prototype has successfully detected both known and unknown viruses. As noted above, signature-based antivirus software detects only previously seen viruses, because in order to have a virus signature in the database, the virus first needs to be analysed. Therefore, traditional antivirus software fails to detect unknown or formerly unseen viruses, leaving systems vulnerable to such viruses. By contrast, despite a number of false negatives among the results reported in Section 7.3.2, our prototype has been shown to have the ability to detect both known and unknown viruses.

Thus, while existing antivirus software succeeds in protecting operating systems from known computer viruses, it fails to deal with novel ones, making it likely

to produce a number of false negatives at any time, given the tens of thousands of new viruses which are said to appear every day and which will not be detected because they are unknown to the antivirus products. By contrast, our prototype has the proven ability to detect not only known viruses but also unknown ones, as long as they behave the same as previously analysed ones, giving it an advantage over traditional signature-based virus detection. In other words, some false negatives are produced by both our detection technique and traditional antivirus software, but the latter fails totally to detect previously unseen viruses, while our approach can detect them.

The second null hypothesis is:

H2: The number of false positives and false negatives when detecting known and unknown computer viruses by their behaviours using AnaTempura **is the same as** or **lower than** the number when detecting known and unknown computer viruses by their signatures.

As our prototype has been found to be successful in detecting both previously seen and unseen viruses, while traditional antivirus products fail in the latter case, the second null hypothesis can be rejected. Therefore, the second alternative hypothesis is supported, because it can be said that the number of false positives and negatives when detecting known and unknown computer viruses by their behaviours using our approach is the same as or lower than the number when detecting known and unknown computer viruses by their signatures.

7.6 Discussion

7.6.1 Dataset Selection

A collection of 289 virus samples, and 50 normal processes that are running in Windows (XP) and Windows 7, were analysed and tested in this research. Because our approach targets Windows 32-bit computer viruses, those analysed and tested were selected from the Win32 category and from the two repositories explained in Chapter 6. The purpose of these repositories, as their websites suggest, is to provide a collection of computer malwares in order to analyse them and protect operating systems from them. Therefore, the classification of these viruses is not very significant or accurate. In other words, a piece of malware may be wrongly categorised as a virus when it is in fact some other type of malware. Thus, some of the malware analysed here may not have been computer viruses and this supports our second justification of apparent false negatives returned by our prototype, as explained in Section 7.3.2.

However, to ensure that the sample size reflects the total number of the population (computer viruses), a little calculation is needed [101]. A number of values are needed to complete the sample size equation:

- **The population Size**

In this research, the population size is the number of computer viruses. However, it is not possible to count the exact number of computer viruses. According to Smith [101], It is common for the population size to be unknown or approximated.

- **Confidence Interval**

The confidence interval or margin of error is needed because it is believed that no sample will be perfect. Therefore, how much error to allow is needed.

- **Confidence Level**

The confidence level determines how the actual mean falls within the chosen confidence interval. The most common confidence intervals are 90% confident, 95% confident, and 99% confident.

- **Standard Deviation**

The standard deviation means how much variance is allowed. Normally, the safe decision is to use (0.5).

The confidence level corresponds to a Z-score which is a constant value needed for the sample size equation. The most common confidence levels of Z-score are as follows:

- 90% – Z Score = 1.645
- 95% – Z Score = 1.96
- 99% – Z Score = 2.326

Next, Z-score, standard deviation, and confidence interval are needed to complete the following equation:

Equation 7.1: Sample Size Equation.

$$Necessary\ Sample\ Size = ((Z\text{-}score)^2 * (\ StdDev * (1\text{-}StdDev))) / (margin\ of\ error)^2$$

The above equation is for an unknown population size or a very large population size [101]. Furthermore, we chose a 90% confidence level, .5 standard

deviation, and a margin of error (confidence interval) of +/- 5%, to be the values needed to complete the sample size equation. Therefore, the following calculation can be deduced:

$$((1.645)^2 \times (.5(.5))) / (.05)^2$$

$$(2.607025 \times .25) / .0025$$

$$.67650625 / .0025$$

$$270.60$$

At least 271 virus samples are needed.

This means that the sample size needed in this research is 271 samples. However, a collection of 289 virus samples, and 50 normal processes were analysed and tested in this research.

7.6.2 Virus Detection

After examining the results of all the samples analysed and tested, two conclusions can be drawn about the theory of attaching a copy of a virus to another file or files. First, as no false positives occurred when the benign processes were tested, it can be said that the theory of attachment is unique to computer viruses and not to normal processes. It can be concluded that the theory of attachment is a characteristic that distinguishes between computer viruses and normal processes. The second conclusion is that while the performance and usability of the operating system may be reduced while the runtime detection is running, this is not an insurmountable obstacle, because our behaviour-based virus detection can be improved to enhance performance and

usability in order to meet the need of end-users to browse the system easily while detection occurs on the fly.

In respect of the false negatives produced by both the analysed and tested viruses, three observations can be made, each having its own unique solution. First, if a candidate virus does not attach itself to another file, it can be considered to be some type of malware other than a virus, as the definition of a virus suggests. Secondly, a computer virus may be expected to do anything; for example, it may build its own kernel API driver and may issue other types of API and Native API calls, helping it to attach itself to other files and thus avoid detection. This would require a better understanding of the low level of the Windows operating system and better programming techniques, which are believed to be accomplishable. The third observation is that these viruses may attach themselves to other files, but our prototype lacks some of the functionality and is not implementable. The best solution to this problem is to complement our approach with other known approaches. It is assumed that the level of true positives will be sustained or increased by the combination, while the number of false negatives will be reduced.

7.7 Limitations

There are two main limitations to the proposed work, concerning other types of malware and the kernel level.

7.7.1 The Kernel Level

The first limitation stems from the characteristics of the kernel level of Microsoft operating systems. It is not easy to program this low level because it is not an open source environment. The rootkit explained in Chapter 5 should give Tempura the Native API calls alongside their information. However, Deviare API can play the role of a rootkit because it can provide the Native API information coming from the kernel level after the call is accepted. On the other hand, Deviare API will not be as accurate as the rootkit because the latter will be at the kernel level and no single Native API can avoid it. This problem amounts to a limitation because it may lead to false negatives.

7.7.2 The Moment of Detection

The second limitation stems from the moment of detection. It can be deduced that our prototype can only detect computer viruses after the file is infected. In other words, after it writes itself to another file. Therefore, two or more files need to be terminated and deleted namely, the original file and the infected ones. Instead, it is better to detect the original virus and deletes it before infecting another file or files. This problem is considered to be a limitation because it may lead to infect the system while the termination is done. This means that, the virus might infect another file before it is terminated.

7.7.3 Solutions

These two limitations should be addressed in future work, as detailed in the next chapter. As to the first limitation, addressing it will require a very deep knowledge of

the low level programming of Microsoft operating systems, which can be accomplished with the help of low level tools that are being developed to improve the understanding of restricted operating systems. The second limitation can be done by looking ahead for API calls, i.e., it will require a very deep knowledge of API calls and how they can be predicted. However, Tempura has the ability and can be programmed to read the API call in advance. Therefore, the virus can be terminated before it completes the comprehensive attempt of attachment.

7.8 Summary

This chapter has discussed in detail the results of the present research concerning both analysed viruses and those tested by our prototype, as well as the benign processes tested by our prototype. These results show that our approach gave no false positives when the normal processes were tested, but that there were a number of false negatives in the case of both analysed and tested viruses. The reasons for these false negatives were discussed in detail.

The prototype was evaluated and compared with existing antivirus products in terms of system performance and usability and of the accurate detection of both known and unknown viruses. The comparison results were discussed in the evaluation section, as was dataset selection. Finally the limitations of the current work were examined and solutions suggested.

These suggestions represent recommendations for future work to study how to analyse more examples of computer malware and combine their behaviours with

those of the viruses presented in this research. It is anticipated that the combination would be able to detect both malware programs attaching themselves to files in local operating system and those which spread across networks. The other proposed line of future work would be to gain a deep knowledge of low level programming to provide accurate information at the kernel level.

The next chapter will present the overall conclusions of this research before discussing in detail the future proposed to solve the limitations identified here.

Chapter 8

Conclusion and Future Work

Objectives:

- Provide a summary of this research.
- Highlight the original contributions to knowledge.
- Revisit the success criteria of this research.
- Highlight the limitations of this research and present the potential future work beyond this thesis.

8.1 Thesis Summary

In this thesis, a general framework that enables specifying, implementing, and validating a behaviour-based virus detection system is developed. Furthermore, from this framework, two architectures emerge, one for the virus analysis that assists in identifying what sort of actions normal and viral programs carry out and how to distinguish between them. API calls, which are used to represent the behaviour of a virus, are extracted using existing tools. In addition, from the analysis architecture, the following five categories with their rules represent virus steps in a behaviour:

- Find to infect
- Get information
- Read and/or copy
- Write and/or delete
- Set information.

The other architecture is the detection one that illustrates how computer viruses can be monitored and detected. Moreover, a parallel execution tool is provided in order to run user and kernel level detection at the same time.

A run-time prototype that simultaneously receives Win32 and Native API calls from both user and kernel levels is developed. Nektra's Deviare API is used to intercept Win32 API calls and deliver them to Tempura at the user level. On the other hand, a 'good' rootkit is also utilised to intercept Native API calls at the kernel levels and transfer them to Tempura. A Java pipe is built to send the information of both calls to Tempura. Finally, Tempura has the ability to provide a final result about a collection of

normal and viral processes based on the five categories with their rules which were previously observed.

A collection of 283 viruses is used in the computer virus analysis part of this research. However, there are a number of false negatives that do not follow the theory of attachment that computer viruses normally carry out. In addition, a number of justifications are provided to vindicate why these false negatives are produced. On the other hand, the prototype's results show that there are no false positives produced by any of the 50 normal processes that are tested by the prototype. In addition, more computer viruses (known and unknown) are tested by the prototype and it has been shown that the prototype in this study has the ability to detect both known and unknown viruses despite the false negative production by some viruses.

An attempt to find a unique characteristic that exists in all computer viruses has been addressed in this research. This characteristic should be able to detect both known and unknown viruses that belong to various categories of computer viruses. Both signature and behaviour-based virus detections are two on-going research areas which have their own problems that require more consideration. In this research, behaviour-based virus detection is based on the theory of a virus attaching itself to another computer file, or files, that is believed to be unique to computer viruses as their definition suggests (see Section 2.3) and not prevalently produced by normal processes. This characteristic was initially formalised using ITL and then programmed with Tempura in order for it to be implemented. As a result, the approach detects any attempt of a process to attach itself to another file or files.

8.2 Contributions

This thesis develops a framework that uses a behaviour-based virus detection system that aims to detect both known and unknown viruses. The contribution of the thesis is as follows:

- A general framework for behaviour-based virus detection system, as shown in Chapter 4.
- One architecture for virus analysis that extracts Win32 and Native API calls of both normal and viral processes in order to understand virus behaviours, as shown in Chapter 4.
- Second architecture for virus detection that uses these calls from both user and kernel levels in order to provide a better detection system and minimise the rates of false negatives and false positives (see Chapter 5).
- A parallel runtime implementation which has the ability to be plugged-in with two tools, namely Nektra's Deviare API (user level), and the kernel rootkit (kernel level), as discussed in Chapter 4.
- Five categories that represent a unique characteristic (the theory of attachment) which leads to detecting both known and unknown viruses that belong to various categories of computer viruses, as present in Chapter 4.
- Four different orders of API calls that a virus normally carries out in order to attach itself to another file (see Chapter 4 and 5).

- A prototype for virus detection that is able to examine more than one process at a time at both levels. This prototype was discussed in Chapter 5.

8.3 Success Criteria Revisited

The success criteria, which have been formulated in order to measure the success of this research, are provided in Chapter 1. These success criteria are revisited in this section in order examine whether they have been met.

- *Detecting known and unknown viruses.*

This is one of the most significant criterion in this research. This criterion will measure the success of the proposed approach. Due to the fact that one of the reasons why behaviour-based virus detection techniques were introduced is to detect unknown viruses, the approach in this research needs to do this. The prototype's results detailed in the previous chapter (see Chapter 7) demonstrate that our prototype is able to detect both known and unknown viruses. This conclusion is drawn due to the 10 specimens (including known and anonymous viruses) examined by the prototype in this study in Section 7.3.2. However, there were a number of false negatives in the case of both analysed and tested viruses. The reasons for these false negatives were discussed in detail in Chapter 7.

- *False positive production.*

  Section 7.3.1 provides the results for the 50 benign processes which were tested by our prototype. As illustrated in Table 7.1, none of these normal

processes attempts to attach itself to another file or files. As a result, it can be said that no false positives were produced by our prototype. This result can lead to a conclusion in which the characteristic of attachment is unique to computer viruses and can be used to distinguish between viral and normal processes.

- *Running with an acceptable system performance.*

  In this research, a system monitor application called Windows Task Manager is used to measure the performance of the two Windows operating systems (Windows XP and 7) which are used for the analysis and testing of sample processes. As demonstrated in Section 7.4, the two operating systems show a normal performance in which the CPU and memory usage were not too busy, despite the few occasions where the computer had to be rebooted. This is considered to be acceptable, since an operating system can run with a very slow performance under normal conditions

- *The ability for users to interact with the operating system while the prototype is running.*

  The final criterion is considered to be significant due to the fact that the end-user needs to enjoy utilising the system without difficulty during the runtime of the prototype. Furthermore, it has been argued in Chapter 7 that users require light and robust virus detection which uses little system memory, so that the user is able to continue using the system while searching for viruses. As demonstrated in Section 7.4, the system resources and programs were

available to the end-user during the runtime of the prototype. Moreover, despite some difficulties such as, the problem of slow running and sometimes the need to reboot the system, different desktop and Internet applications can be used by the end-user during the runtime detection.

8.4 Limitations

The following points are the limitations of the proposed research. Most of the limitations are, in fact, practical limitations that refer to the implementation of this research or the nature of computer viruses.

- It has been demonstrated that the programming of the Windows kernel level is a very hard task which requires very extensive knowledge of the low level programming of Microsoft operating systems. Therefore, the Deviare API tool plays the role of the kernel rootkit (discussed in Chapter 5) in which Deviare API can intercept the Native API calls coming from the kernel. This is considered to be a limitation due to the fact that the rootkit is at the kernel level and no single Native API can avoid it, however, Deviare API might be avoided.

- It has been shown that the theory of attachment is unique to computer viruses. On the other hand, the moment of detecting the attempt of a computer virus to attach itself to a file should be before the attempt is carried out. In addition, the prototype detection of this research can only

detect a computer virus after it writes itself to another file. Therefore, this late detection is considered to be one of the limitations of the research.

8.5 Future Work

The literature review in this thesis demonstrates that the area of computer virus detection is a challenging area due to the fact that it is a never-ending fight between attackers and defenders. Behaviour-based virus detection techniques are especially interesting because they overcome the problems associated with traditional signature-based detection. As suggested by [1, 9], the nature of computer viruses leads virus researchers to seek alternative solutions. Therefore, future research for any virus detection approach is needed in order to enhance it.

Furthermore, the following points are some proposals for future research:

- Analysing other types of computer malware to examine how they act within the operating system and networks. This task is likely to require a research team to solve it as it requires much analysis of many types of malware to determine their behaviour.

- A deep knowledge of the low level programming of Microsoft operating systems that would help us to build the rootkit and develop it to be a part of the OS as a device driver. This rootkit should have the ability to provide Native API calls at the kernel level, thus resulting in our system being more reliable.

- A deep knowledge of API calls and how they can be intercepted in advance. This task will give our approach the ability to detect the attempt of attachment before the target file is infected.

- As explained in Chapter 2, the problems associated with the heuristic-based virus detection can be solved by using the behaviour-based virus detection. Therefore, one the future research will be to investigate the use of our research findings to solve these problems.

Bibliography

[1]. SZOR, P., 2005. *The art of computer virus research and defense.* Addison-Wesley Professional.

[2]. LEON, M., 7th Jan 2000. Internet virus boom. *InfoWorld,* pp. 36-37.

[3]. HARMER, P.K., WILLIAMS, P.D., GUNSCH, G.H. and LAMONT, G.B., 2002. An artificial immune system architecture for computer security applications. *Evolutionary Computation, IEEE Transactions on,* 6(3), pp. 252-280.

[4]. FILIOL, E., 2005. *Computer viruses: from theory to applications.* Springer Paris etc.

[5]. DAVIS, M., BODMER, S. and LEMASTERS, A., 2009. *Hacking Exposed Malware and Rootkits.* McGraw-Hill, Inc.

[6]. NOWICKA, E. and ZAWADA, M., 2006. Modeling Temporal Properties of Multi-event Attack Signatures in Interval Temporal Logic, *Proceedings of the IEEE/IST Workshop on Monitoring, Attack Detection and Mitigation(MonAM 2006) Tuebingen, Germany* 2006.

[7]. KASPERSKY, 2013-last update, How to keep your antivirus software up-to-date [Homepage of Kaspersky], [Online]. Available: http://www.kaspersky.co.uk/threats/update-your-antivirus-software [04/06, 2011].

[8]. , Spyware Removal - Virus Removal | Norton Premium Services2013-last update. Available: http://us.norton.com/support/premium_services/virusfaq.jsp [7/9/2013].

[9]. BRITT, W., GOPALASWAMY, S., HAMILTON, J.A., DOZIER, G.V. and CHANG, K.H., 2007. Computer defense using artificial intelligence, *Proceedings of the 2007 spring simulation multiconference-Volume 3* 2007, Society for Computer Simulation International, pp. 378-386.

[10]. SONG, I., LEE, Y. and KWON, T., 2005. A multi-gigabit virus detection algorithm using ternary CAM. *Computational Intelligence and Security,* pp. 220-227.

[11]. KINDER, J., KATZENBEISSER, S., SCHALLHART, C. and VEITH, H., 2005. Detecting malicious code by model checking. *Detection of Intrusions and Malware, and Vulnerability Assessment,* pp. 514-515.

[12]. LEEDY, P.D. and ORMROD, J.E., 2010. *Practical research: planning and design.* Upper Saddle River, NJ: Merrill.

[13]. BURNS, R., 2000. *Introduction to research methods.* 4th edition. Sage London.

[14]. DANTZKER, M.L. and HUNTER, R.P., 2011. *Research Methods for Criminology and Criminal Justice.* Jones & Bartlett Learning.

[15]. COHEN, F., 1987. Computer viruses: theory and experiments. *Computers & Security,* 6(1), pp. 22-35.

[16]. COHEN, F.B. and COHEN, D.F., 1994. *A short course on computer viruses.* John Wiley & Sons, Inc.

[17]. WANG, J., 2009. *Computer network security: theory and practice.* Springer.

[18]. BOYCE, J., 2011. *Windows 7 bible.* Wiley.

[19]. HEVNER, A. and CHATTERJEE, S., 2010. *Design Research in Information Systems: Theory and Practice.* Springer.

[20]. ZELTSER, L., 2013-last update, 5 Steps to Building a Malware Analysis Toolkit Using Free Tools. Available: http://zeltser.com/malware-analysis-toolkit/ [04/06, 2011].

[21]. Home EICAR - European Expert Group for IT-Security. Available: http://www.eicar.org/56-0-Home.html [3/3/2013, 2013].

[22]. ALAZAB, M., VENKATARAMAN, S. and WATTERS, P., 2010. Towards Understanding Malware Behaviour by the Extraction of API Calls, *Second Cybercrime and Trustworthy Computing Workshop* 2010, pp. 52-59.

[23]. MORALES, J.A., CLARKE, P.J. and DENG, Y., 2010. Identification of file infecting viruses through detection of self-reference replication. *Journal in computer virology,* 6(2), pp. 161-180.

[24]. KASPERSKY, E., 2006-last update, Problems for AV vendors: Some thoughts [Homepage of Kaspersky Lab, Russia], [Online]. Available: http://www.virusbtn.com/virusbulletin/archive/2006/04/vb200604-comment.dkb?mobile_on=yes [01/31, 2011].

[25]. EVERS, J., January 19, 2006, 2006-last update, Computer crimes cost 67 billion, FBI says [Homepage of Cnet], [Online]. Available: http://news.cnet.com/2100-7349_3-6028946.html [01/31, 2011].

[26]. SIDDIQUI, M.A., 2008. *Data mining methods for malware detection.* ProQuest.

[27]. SKOUDIS, E. and ZELTSER, L., 2004. *Malware: Fighting malicious code.* Prentice Hall PTR.

[28]. ADLEMAN, L., 1990. An abstract theory of computer viruses, *Advances in Cryptology—CRYPTO'88* 1990, Springer, pp. 354-374.

[29]. MORALES, J.A., 2008. *A behavior based approach to virus detection*, Florida International University.

[30]. SINGH, P.K. and LAKHOTIA, A., 2002. Analysis and detection of computer viruses and worms: An annotated bibliography. *SIGPLAN Notices,* 37(2), pp. 29-35.

[31]. RABAH, K., 2005. Secure implementation of message digest, authentication and digital signature. *Information Technology Journal,* 4(3), pp. 204-221.

[32]. YOO, I.S. and ULTES-NITSCHE, U., 2006. Non-signature based virus detection. *Journal in Computer Virology,* 2(3), pp. 163-186.

[33]. LIVINGSTON, B., 23/02/2006, 2006-last update, How Long Must You Wait for an Anti-Virus Fix? - eSecurity Planet. Available: http://www.esecurityplanet.com/views/article.php/3316511/How-Long-Must-You-Wait-for-an-AntiVirus-Fix.htm [2/2/2013].

[34]. CHRISTODORESCU, M., JHA, S., MAUGHAN, D., SONG, D. and WANG, C., 2006. *Malware Detection.* Springer.

[35]. CONRY-MURRAY, A., 2002. Behavior-blocking stops unknown malicious code. *Network Magazine*.

[36]. MESSMER, E., 01/28/02, 2002-last update, Behavior blocking repels new viruses [Homepage of Network World Fusion], [Online]. Available: http://www.networkworld.com/news/2002/0128antivirus.html [02/02/2011].

[37]. ELLIS, D.R., AIKEN, J.G., ATTWOOD, K.S. and TENAGLIA, S.D., 2004. A behavioral approach to worm detection, *Proceedings of the 2004 ACM workshop on Rapid malcode* 2004, ACM, pp. 43-53.

[38]. CHIANG, H. and TSAUR, W., 2010. Mobile Malware Behavioral Analysis and Preventive Strategy Using Ontology, *Social Computing (SocialCom), 2010 IEEE Second International Conference on* 2010, IEEE, pp. 1080-1085.

[39]. IDIKA, N. and MATHUR, A.P., 2007. A survey of malware detection techniques. *Purdue University,* pp. 48.

[40]. ZHANG, Q., 2008. *Polymorphic and metamorphic malware detection.* ProQuest.

[41]. SKORMIN, V.A., 2010. Server Level Analysis of Network Operation Utilizing System Call Data. *BINGHAMTON UNIV NEW YORK DEPT OF ELECTRICAL AND COMPUTER ENGINEERING*.

[42]. BOS, H., 2013-last update, D16 (D4. 2) Analysis Report of Behavioral Features [Homepage of Wombat], [Online]. Available: http://www.wombat-project.eu/WP4/FP7-ICT-216026-Wombat_WP4_D16_V01_Analysis-Report-of-Behavioral-features.pdf [12/20/2012].

[43]. MOSKOVITCH, R., ELOVICI, Y. and ROKACH, L., 2008. Detection of unknown computer worms based on behavioral classification of the host. *Computational Statistics & Data Analysis,* 52(9), pp. 4544-4566.

[44]. ALTAHER, A., RAMADASS, S. and ALI, A., 2011. Computer virus detection using features ranking and machine learning. *Australian Journal of Basic and Applied Sciences,* 5(9), pp. 1482-1486.

[45]. SKORMIN, V., VOLYNKIN, A., SUMMERVILLE, D. and MORONSKI, J., 2007. Prevention of information attacks by run-time detection of self-replication in computer codes. *Journal of Computer Security,* 15(2), pp. 273-302.

[46]. VEERAMANI, R. and RAI, N., 2012. Windows API based Malware Detection and Framework Analysis. *International Journal of Scientific & Engineering Research (IJSER),* 3(3).

[47]. RAVI, C. and MANOHARAN, R., 2012. Malware Detection using Windows API Sequence and Machine Learning. *International Journal of Computer Applications,* 43(17), pp. 12-16.

[48]. SEIFERT, C., STEENSON, R., WELCH, I., KOMISARCZUK, P. and ENDICOTT-POPOVSKY, B., 2007. Capture–A behavioral analysis tool for applications and documents. *digital investigation,* 4, pp. 23-30.

[49]. RUSSINOVICH, M., 2011-last update, Inside the Native API [Homepage of Sysinternals], [Online]. Available: http://www.sysinternals.com/Information/NativeApi.html [1/22/2011].

[50]. RESCUE, D., 2006. IDA Pro Disassembler. *2006-10-20).*http://www.datarescue.com/idabase.

[51]. ZWANGER, V. and FREILING, F.C., 2013. Kernel mode API spectroscopy for incident response and digital forensics, *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop* 2013, ACM, pp. 3.

[52]. BAYER, U., MOSER, A., KRUEGEL, C. and KIRDA, E., 2006. Dynamic analysis of malicious code. *Journal in Computer Virology,* 2(1), pp. 67-77.

[53]. JACOB, G., DEBAR, H. and FILIOL, E., 2008. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology,* 4(3), pp. 251-266.

[54]. LUO, J., LUO, G. and ZHAO, Y., 2011. Satisfiability degree computation for linear temporal logic, *Cognitive Informatics & Cognitive Computing (ICCI* CC), 2011 10th IEEE International Conference on* 2011, IEEE, pp. 373-380.

[55]. VARDI, M., 2009. From philosophical to industrial logics. *Logic and Its Applications,* pp. 89-115.

[56]. CHENA, Q., SUD, K., WUE, L. and XUB, Z., 2012. Model Checking Cooperative Multi-agent Systems in BDI Logic. *Journal of Information & Computational Science,* 9:(5), pp. 1185-1194.

[57]. CLARKE, E.M., GRUMBERG, O. and PELED, D.A., 2000. *Model checking.* MIT press.

[58]. HUTH, M. and RYAN, M., 2004. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press Cambridge, UK.

[59]. ESHUIS, R., 2001. Model checking activity diagrams in TCM. *Technical report, University of Twente,* .

[60]. MOSZKOWSKI, B., 1984. *Executing temporal logic programs.* 55. Computer Laboratory, University of Cambridge: .

[61]. CAU, A., MOSZKOWSKI, B. and ZEDAN, H., 2009. Interval temporal logic. *URL:* http://www.cms.dmu.ac.uk/~cau/itlhomepage/itlhomepage.html.

[62]. SIEWE, F., 2005. *A compositional framework for the development of secure access control systems*, De Montfort University.

[63]. ZHOU, S., ZEDAN, H. and CAU, A., 2005. Run-time analysis of time-critical systems. *Journal of Systems Architecture,* 51(5), pp. 331-345.

[64]. YE, N., 2000. A markov chain model of temporal behavior for anomaly detection, *Proceedings of the 2000 IEEE Systems, Man, and Cybernetics Information Assurance and Security Workshop* 2000, Oakland: IEEE, pp. 169.

[65]. CHANDOLA, V., BANERJEE, A. and KUMAR, V., 2012. Anomaly detection for discrete sequences: A survey. *Knowledge and Data Engineering, IEEE Transactions on,* 24(5), pp. 823-839.

[66]. SINGH, P.K. and LAKHOTIA, A., 2003. Static verification of worm and virus behavior in binary executables using model checking, *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society* 2003, IEEE, pp. 298-300.

[67]. SONG, F. and TOUILI, T., 2012. Pushdown model checking for malware detection. *Tools and Algorithms for the Construction and Analysis of Systems,* pp. 110-125.

[68]. HOLZER, A., KINDER, J. and VEITH, H., 2007. Using verification technology to specify and detect malware. *Computer Aided Systems Theory–EUROCAST 2007,* pp. 497-504.

[69]. NALDURG, P., SEN, K. and THATI, P., 2004. A temporal logic based framework for intrusion detection. *Formal Techniques for Networked and Distributed Systems–FORTE 2004,* pp. 359-376.

[70]. MUÑOZ, A., GONZALEZ, J. and MAÑA, A., 2012. A Performance-Oriented Monitoring System for Security Properties in Cloud Computing Applications. *The Computer Journal,* 55 (8), pp. 979-994.

[71]. Overview of the Windows API (Windows)3/25/2010, 2010-last update [Homepage of Microsoft], [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/apps/aa383723(v=vs.85).aspx [23/1/2011].

[72]. WIKIPEDIA CONTRIBUTORS, , Wikipedi [Homepage of Wikipedia, The Free Encyclopedia], [Online]. Available: http://en.wikipedia.org [2013].

[73]. CHRISTODORESCU, M. and JHA, S., 2006. Static analysis of executables to detect malicious patterns. *WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES,* .

[74]. CHRISTODORESCU, M., JHA, S., SESHIA, S.A., SONG, D. and BRYANT, R.E., 2005. Semantics-aware malware detection, *Security and Privacy, 2005 IEEE Symposium on* 2005, IEEE, pp. 32-46.

[75]. KRUEGEL, C., ROBERTSON, W. and VIGNA, G., 2004. Detecting kernel-level rootkits through binary analysis, *Computer Security Applications Conference, 2004. 20th Annual* 2004, IEEE, pp. 91-100.

[76]. *Oracle VM VirtualBox.* https://www.virtualbox.org/.

[77]. ROBIN, J.S. and IRVINE, C.E., 2000. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. *NAVAL POSTGRADUATE SCHOOL MONTEREY CA DEPT OF COMPUTER SCIENCE*.

[78]. RUTKOWSKA, J., 2004. Red Pill: how to detect VMM using (almost) one CPU instruction. *Invisible Things*.

[79]. PIZZONIA, M. and RIMONDINI, M., 2008. Netkit: easy emulation of complex networks on inexpensive hardware, *Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities*

2008, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 7.

[80]. SHARIF, M., YEGNESWARAN, V., SAIDI, H., PORRAS, P. and LEE, W., 2008. Eureka: A framework for enabling static malware analysis. *Computer Security-ESORICS 2008,* , pp. 481-500.

[81]. DINABURG, A., ROYAL, P., SHARIF, M. and LEE, W., 2008. Ether: malware analysis via hardware virtualization extensions, *Proceedings of the 15th ACM conference on Computer and communications security* 2008, ACM, pp. 51-62.

[82]. KANG, M.G., POOSANKAM, P. and YIN, H., 2007. Renovo: A hidden code extractor for packed executables, *Proceedings of the 2007 ACM workshop on Recurring malcode* 2007, ACM, pp. 46-53.

[83]. SNAKER, QWERTON, JIBZ, AND XINEOHP, 2011-last update, *PEiD*. Available: http://www.peid.info/ [20/6/2011].

[84]. Blade API Monitor - Home. http://www.bladeapimonitor.com/.2008.

[85]. ROHITAB, *API Monitor.* http://www.rohitab.com/apimonitor: Rohitab.

[86]. *VX Heavens*2011-last update. Available: http://vx.netlux.org [5/13/2010].

[87].Offensive Computing, 2013-last update. Available: http://www.offensivecomputing.net [5/20/2010].

[88]. HOGLUND, G. and BUTLER, J., 2006. *Rootkits: subverting the Windows kernel.* Addison-Wesley Professional.

[89]. RIES, C., 2006. Inside windows rootkits. *VigilantMinds Inc,* 4736.

[90]. MOSZKOWSKI, B., 1993. *Some very compositional temporal properties.* 466. Dept. of Computing Science, University of Newcastle: .

[91]. MARHUSIN, M.F., LARKIN, H., LOKAN, C. and CORNFORTH, D., 2008. An evaluation of api calls hooking performance, *Computational Intelligence and Security, 2008. CIS'08. International Conference on* 2008, IEEE, pp. 315-319.

[92]. NEKTRA, 2007-last update, Deviare API Hook [Homepage of http://www.nektra.com/products/deviare], [Online] [1/10/2012].

[93]. CreateFile function (Windows), 2013-last update. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx [6/21/2012].

[94]. CurrPorts: Monitoring TCP/IP network connections on Windows. Available: http://www.nirsoft.net/utils/cports.html [3/7/2013, 2013].

[95]. GRAVES, K., 2007. *CEH: OFFICIAL CERTIFIED ETHICAL HACKER REVIEW GUIDE (With CD).* John Wiley & Sons.

[96]. Kaspersky Lab UK :: Antivirus software, 2013-last update. Available: http://www.kaspersky.co.uk/ [2/20/2012].

[97]. Computer's performance using Task Manager, 2012-last update. Available: http://windows.microsoft.com/en-US/windows-vista/See-details-about-your-computers-performance-using-Task-Manager [9/20/2012].

[98]. McAfee—Antivirus, Encryption, Firewall, Email Security, Web Security, Risk & Compliance. Available: http://www.mcafee.com/uk/ [12/20/2012].

[99]. YUAN, F., 2001. *Windows Graphics Programming: Win32 GDI and DirectDraw.* Prentice Hall.

[100]. Microsoft Security Essentials - Microsoft Windows. Available: http://windows.microsoft.com/en-US/windows/security-essentials-download [12/20, 2012].

[101]. SCOTT, S., *Determining Sample Size: How to Ensure You Get the Correct Sample Size.* http://www.qualtrics.com/blog/determining-sample-size/ edn. Qualtrics Blog. 2013.

Appendix A

Experiments Result

1. Analysis result

The results of the analysis of 283 viruses using existing tools, as explained in Chapter 4, are shown in Tables A.1 and A.2. The names of the viruses analysed are listed in the first and third columns, while the second and fourth columns list the results of testing the theory of attachment (five steps for a virus to attach itself to another file). These results indicate whether or not each virus attempted to attach itself by following the five categories. Thus, “Yes” means that the virus or malware issued Native or API calls related to the five categories with their rules, while “No” means that it did not.

Table A.1: Results of virus analysis-1.

Process Name	Attachment	Process Name	Attachment
Malware.Win32/Alcra	Yes	Malware.Win32/Bropia	Yes
Malware.Win32/Allaple	Yes	Malware.Win32/BugBear	Yes

Malware.Win32/Areses	Yes	Malware.Win32/Chir	Yes
Malware.Win32/Bagle	Yes	Malware.Win32/Cabanas	No
Malware.Win32/Evaman	Yes	Malware.Win32/Mugly	Yes
Malware.Win32/Gibe	Yes	Malware.Win32/Poebot	Yes
Malware.Win32/Tutiam	Yes	Malware.Win32/VB.CA	Yes
Malware.Win32/Rbot	No	Malware.Win32/Mytob	Yes
Malware.Win32/Rontokbro	Yes	Malware.Win32/Sober	Yes
Malware.Win32/Locksky	Yes	Malware.Win32/Elkern	No
Malware.Win32/Korgo	Yes	Malware.Win32/Flopcopy	Yes
Malware.Win32/Zafi	Yes	Malware.Win32/Vanbot	Yes
Malware.Win32/Vote	No	Malware.Win32/Stration	Yes
Malware.Win32/Sobig	Yes	Malware.Win32/Sohanad	Yes
Malware.Win32/Ska	Yes	Malware.Win32/Skudex	Yes
Malware.Win32/Redesi	Yes	Malware.Win32/Paukor	Yes
Malware.Win32/Neveg	Yes	Malware.Win32/Netsky	Yes
Malware.Win32/Myparty	Yes	Malware.Win32/Moonlight	Yes
Malware.Win32/Minusi	Yes	Malware.Win32/Mapson	Yes
Malware.Win32/Lovgate	Yes	Malware.Win32/Looked	Yes
Malware.Win32/Lioten	Yes	Malware.Win32/Holar	Yes
Malware.Win32/Hocgaly	Yes	Malware.Win32/Higuy	Yes
Malware.Win32/Gnuman	Yes	Malware.Win32/Frethem	Yes
Malware.Win32/Fix2100	Yes	Malware.Win32/ExploreZip	Yes
Malware.Win32/Gain	Yes	Malware.Win32/Lirva	Yes
Malware.Win32/Blaster	Yes	Malware.Win32/Hybris	Yes
Malware.Win32/Antiman	Yes	Malware.Win32/Cloner	Yes
Malware.Win32/Cervivec	Yes	Malware.Win32/Tenga	Yes
Malware.Win32/Badtrans	No	Malware.Win32/Doomjuice	Yes
Malware.Win32/Hai	Yes	Malware.Win32/Funner	Yes
Malware.Win32/Reatle	Yes	Malware.Win32/Rinbot	Yes
Malware.Win32/Polip	No	Malware.Win32/Puce	Yes
Malware.Win32/Parite	Yes	Malware.Win32/Qaz	Yes
Malware.Win32/Nachi	Yes	Malware.Win32/Mywife	Yes
Malware.Win32/Magistr	Yes	Malware.Win32/Maslan	Yes
Malware.Win32/Klez	Yes	Malware.Win32/Kipis	Yes
Malware.Win32/Kidala	Yes	Malware.Win32/Gurong	Yes
Malware.Win32/Funlove	Yes	Malware.Win32/Padobot	Yes
Malware.Win32/Bozori	Yes	Malware.Win32/Bofra	Yes
Malware.Win32/Sality	Yes	Malware.Win32/Anzae	Yes
Malware.Win32/Golten	Yes	Malware.Win32/Myfip	Yes
Malware.Win32/Philis	Yes	Malware.Win32/Theals	No
Malware.Win32/Tirbot	Yes	Malware.Win32/Savage	Yes
Malware.Win32/Autex	Yes	Malware.Win32/Backterra	No
Malware.Win32/Deborm	Yes	Malware.Win32/Dedler	Yes
Malware.Win32/Cone	Yes	Malware.Win32/Chimo	Yes
Malware.Win32/Fanbot	Yes	Malware.Win32/Floppy	Yes
Malware.Win32/Heretic	Yes	Malware.Win32/Hotlix	Yes
Malware.Win32/Zusha	Yes	Malware.Win32/Jupir	Yes
Malware.Win32/Kelvir	Yes	Malware.Win32/Kindal	Yes

Malware.Win32/MTX-m	Yes	Malware.Win32/MyLife	Yes
Malware.Win32/Snapper	Yes	Malware.Win32/Zindos	Yes
Malware.Win32/Annil	Yes	Malware.Win32/Antiqfx	Yes
Malware.Win32/Onamu	Yes	Malware.Win32/PrettyPark	Yes
Malware.Win32/Xddtray	Yes	Malware.Win32/Maddis	Yes
Malware.Win32/Apsiv	No	Malware.Win32/Benjamin	Yes
Malware.Win32/Choke	Yes	Malware.Win32/Dabber	Yes
Malware.Win32/Dipnet	Yes	Malware.Win32/Donk	Yes
Malware.Win32/Gregcenter	Yes	Malware.Win32/Imbiat	Yes
Malware.Win32/HLLP.DeTroie	Yes	Malware.Win32/Kelino	Yes
Malware.Win32/Logpole	Yes	Malware.Win32/Loxar	Yes
Malware.Win32/Mellon	Yes	Malware.Win32/Misodene	Yes
Malware.Win32/Neklace	Yes	Malware.Win32/Pepex	Yes
Malware.Win32/RAHack	Yes	Malware.Win32/Randin	Yes
Malware.Win32/Rirc	Yes	Malware.Win32/Stator	Yes
Malware.Win32/Tumbi	Yes	Malware.Win32/Datom	Yes
Malware.Win32/Tzet	No	Malware.Win32/Zar	Yes
Malware.Win32/Unfunner	Yes	Malware.Win32/Yanz	Yes
Malware.Win32/Upering	Yes	Malware.Win32/Wozer	Yes
Malware.Win32/Vavico	Yes	Malware.Win32/Warpigs	No
Malware.Win32/Visilin	Yes	Malware.Win32/Wallz	Yes

Table A.2: Results of virus analysis-2.

Process Name	Attachment	Process Name	Attachment
Malware.Win32/Codbot	Yes	Malware.Win32/Eliles	Yes
Malware.Win32/Detnat	Yes	Malware.Win32/Eyeveg	Yes
Malware.Win32/Darby	Yes	Malware.Win32/Feebs	Yes
Malware.Win32/Dumaru	Yes	Malware.Win32/Forbot	Yes
Malware.Win32/Fizzer	Yes	Malware.Win32/Fujacks	Yes
Malware.Win32/Mydoom	Yes	Malware.Win32/Wukill	Yes
Malware.Win32/Spybot	Yes	Malware.Win32/Sdbot	Yes
Malware.Win32/Oddbob	Yes	Malware.Win32/Agobot	Yes
Malware.Win32/Mocbot	Yes	Malware.Win32/Mimail	Yes
Malware.Win32/Bobax	Yes	Malware.Win32/Zotob	Yes
Malware.Win32/Aimbot	Yes	Malware.Win32/Yaha	Yes
Malware.Win32/Wootbot	Yes	Malware.Win32/Virut	Yes
Malware.Win32/Pesin	Yes	Malware.Win32/Small	No
Malware.Win32/Sircam	No	Malware.Win32/Sixem	Yes
Malware.Win32/Ritdoor	Yes	Malware.Win32/Reper	Yes
Malware.Win32/Oror	Yes	Malware.Win32/Outa	Yes
Malware.Win32/Mytobor	Yes	Malware.Win32/Mypics	Yes
Malware.Win32/Monikey	Yes	Malware.Win32/Mobler	Yes
Malware.Win32/Maldal	Yes	Malware.Win32/Melissa	Yes
Malware.Win32/Lolol	Yes	Malware.Win32/Satir	Yes
Malware.Win32/Gokar	Yes	Malware.Win32/Hantaner	Yes
Malware.Win32/Goner	Yes	Malware.Win32/Heidi	Yes

Malware.Win32/Fbound	Yes	Malware.Win32/Banwarum	Yes
Malware.Win32/Beast	Yes	Malware.Win32/Antiax	Yes
Malware.Win32/Blebla	Yes	Malware.Win32/Apost	Yes
Malware.Win32/Ircbot	Yes	Malware.Win32/Appflet	Yes
Malware.Win32/Torvil	Yes	Malware.Win32/Traxg	Yes
Malware.Win32/Valla	Yes	Malware.Win32/Qeds	No
Malware.Win32/Womble	Yes	Malware.Win32/Serflog	Yes
Malware.Win32/Flukan	No	Malware.Win32/Fatcat	Yes
Malware.Win32/Sasser	Yes	Malware.Win32/Plexus	Yes
Malware.Win32/Rants	Yes	Malware.Win32/Opaserv	Yes
Malware.Win32/Nugache	Yes	Malware.Win32/Nimda	Yes
Malware.Win32/Mabutu	Yes	Malware.Win32/Luder	Yes
Malware.Win32/Lovelorn	Yes	Malware.Win32/Kriz	Yes
Malware.Win32/Jeefo	Yes	Malware.Win32/Kebede	Yes
Malware.Win32/Inforyou	Yes	Malware.Win32/Ganda	Yes
Malware.Win32/Nanspy	Yes	Malware.Win32/Cissi	Yes
Malware.Win32/Bagz	Yes	Malware.Win32/Atak	Yes
Malware.Win32/Delf	Yes	Malware.Win32/Braid	Yes
Malware.Win32/Opanki	No	Malware.Win32/Aliz	No
Malware.Win32/Tenrobot	Yes	Malware.Win32/Swen	Yes
Malware.Win32/Pinom	Yes	Malware.Win32/Assasin	Yes
Malware.Win32/Deloder	Yes	Malware.Win32/Capside	Yes
Malware.Win32/Derdero	Yes	Malware.Win32/Doep	Yes
Malware.Win32/Bube	Yes	Malware.Win32/Drefir	No
Malware.Win32/Guap	Yes	Malware.Win32/Harwig	Yes
Malware.Win32/HPS	Yes	Malware.Win32/Tibick	Yes
Malware.Win32/Kalel	Yes	Malware.Win32/Kassbot	Yes
Malware.Win32/Aplore	Yes	Malware.Win32/Licu	Yes
Malware.Win32/Raleka	No	Malware.Win32/Randex	Yes
Malware.Win32/Ahker	Yes	Malware.Win32/Anap	Yes
Malware.Win32/Cuebot	Yes	Malware.Win32/Deadcode	Yes
Malware.Win32/Primat	Yes	Malware.Win32/Protoride	No
Malware.Win32/Mofei	Yes	Malware.Win32/Antinny	Yes
Malware.Win32/Bereb	Yes	Malware.Win32/Bilay	Yes
Malware.Win32/Darker	Yes	Malware.Win32/Buchon	Yes
Malware.Win32/Faisal	No	Malware.Win32/Francette	No
Malware.Win32/Jared	Yes	Malware.Win32/ Jitux	Yes
Malware.Win32/Krepper	No	Malware.Win32/Lacrow	Yes
Malware.Win32/LyndEgg	Yes	Malware.Win32/Magold	Yes
Malware.Win32/Mona	Yes	Malware.Win32/Navidad	Yes
Malware.Win32/PMX	Yes	Malware.Win32/Qizy	Yes
Malware.Win32/Reur	Yes	Malware.Win32/Salga	Yes
Malware.Win32/Tanked	Yes	Malware.Win32/Titog	Yes
Malware.Win32/Allocup	Yes	Malware.Win32/Amus	Yes
Malware.Win32/Envid	Yes	Malware.Win32/Shuck	Yes
Malware.Win32/Looksky	Yes	Malware.Win32/Smibag	No
Malware.Win32/Semapi	Yes	Malware.Win32/Smeagol	Yes
Malware.Win32/Seppuku	No	Malware.Win32/Silva	Yes

Malware.Win32/Shodabot	Yes	[redacted]

2. Prototypes's results

As explained in Chapter 6 and 7, the prototype's results are divided into two parts, namely, normal processes' results and computer viruses' results. However, due to the big size of the log files of both normal and viral processes, selected ones will be provided here. Firstly, the normal processes results will be examined.

❖ Normal Processes

Example 1:

This example considers both *chrome.exe* and *firefox.exe* normal processes which were examined at the same time. As shown in the below snippet, It can be seen that both processes issue a *ReadFile* API calls which represent category three in this research but never issue calls from the previous categories. Moreover, *firefox.exe* issues Win32 and Native API calls from the third (*ReadFile*), fourth (*NtSetInformationFile*), and fifth (*WriteFile*) categories respectively. These sequences of API calls resemble the ones which are normally issued by computer viruses but as can be seen from the log file below, the whole order is not complete and then this process is considered to be a normal process.

Listing A.1: Snapshot of firefox.exe and chrome.exe log file.

```
State 1596: Call="CloseHandle"
State 1596: S=0
```

State 1596: ProcessName="chrome.exe"

State 1596: Call="CloseHandle"

State 1596: S=0

State 1596: ProcessName="chrome.exe"

State 1598: Call="NtSetInformationFile"

State 1598: S=0

State 1598: ProcessName="firefox.exe"

State 1598: Call="NtSetInformationFile"

State 1598: S=0

State 1598: ProcessName="firefox.exe"

State 1600: Call="NtClose"

State 1600: S=0

State 1600: ProcessName="chrome.exe"

State 1600: Call="NtClose"

State 1600: S=0

State 1600: ProcessName="chrome.exe"

State 1602: Call="ReadFile" *

State 1602: S=0

State 1602: ProcessName="firefox.exe"

State 1602: Call="ReadFile" *

State 1602: S=0

State 1602: ProcessName="firefox.exe"

State 1604: Call="NtReadFile"

State 1604: S=0

State 1604: ProcessName="firefox.exe"

State 1604: Call="NtReadFile"

State 1604: S=0

State 1604: ProcessName="firefox.exe"

State 1606: Call="NtSetInformationFile"

State 1606: S=0

State 1606: ProcessName="firefox.exe"

State 1606: Call="NtSetInformationFile"

State 1606: S=0

State 1606: ProcessName="firefox.exe"

State 1608: Call="ReadFile" *

State 1608: S=0

State 1608: ProcessName="firefox.exe"

State 1608: Call="ReadFile"

State 1608: S=0

State 1608: ProcessName="firefox.exe"

State 1610: Call="NtReadFile"

State 1610: S=0

State 1610: ProcessName="firefox.exe"

State 1610: Call="NtReadFile"

State 1610: S=0

State 1610: ProcessName="firefox.exe"

```
State 1612: Call="NtSetInformationFile"                    *
State 1612: S=0
State 1612: ProcessName="firefox.exe"
State 1612: Call="NtSetInformationFile"
State 1612: S=0
State 1612: ProcessName="firefox.exe"
State 1614: Call="WriteFile"                               *
State 1614: S=0
State 1614: ProcessName="firefox.exe"
State 1614: Call="WriteFile"
State 1614: S=0
State 1614: ProcessName="firefox.exe"
```

Example 2:

In this example the benign processes *cisvc.exe* and *lsass.exe* are also tested. Snapshots of the log files of both processes are provided below in Listing A.2. It can be observed that both processes issue calls from different categories but on the other hand they do not complete the whole attempt of attaching themselves to other files (the five categories with their rules) that viruses usually carry out. As shown below, *cisvc.exe* issue calls from category one and two but never follow the theory of attachment and issues calls from all the five categories. Hence, this process will not be considered as a viral process.

Listing A.2: Snapshot of lsass.exe and cisvc.exe log file.

```
State 122: Call="NtWriteFile"

State 122: S=0

State 122: ProcessName="lsass.exe"

State 122: Call="NtWriteFile"

State 122: S=0

State 122: ProcessName="lsass.exe"

State 124: Call="NtReadFile"

State 124: S=0

State 124: ProcessName="lsass.exe"

State 124: Call="NtReadFile"

State 124: S=0

State 124: ProcessName="lsass.exe"

State 394: Call="NtQueryDirectoryFile"

State 394: S=0

State 394: ProcessName="cisvc.exe"

State 394:  PID:760 issued a call in cat1                    *

State 394: Call="NtQueryDirectoryFile"

State 394: S=0

State 394: ProcessName="cisvc.exe"

State 1834: Call="GetFileSize"

State 1834: S=1

State 1834: ProcessName="cisvc.exe"
```

```
State 1834:  PID:760 issued a call in cat2                    *
State 1834: Call="GetFileSize"
State 1834: S=0
State 1834: ProcessName="cisvc.exe"
```

❖ Computer Viruses

Example 1:

In this example the computer virus *Weakas.exe* will be tested. This sample is a Windows 32 computer virus. As shown in the snapshot below, this virus has attached itself to another file by following the order of (Cat2→ Cat1→ Cat3→ Cat4→ Cat5). As a result, this virus has been detected by our prototype. In addition, the order that *Weakas.exe* virus follows in order to infect another file is shown in Figure A.1.

Listing A.3: Snapshot of Weakas.exe log file (after detection).

```
State  15: Call="GetFileAttributesW"
State  15: S=0
State  15: ProcessName="Weakas.exe"
State  15:  PID:2132 issued a call in cat2                    *
State  16: Call="NtQueryAttributesFile"
State  16: S=7
State  19: Call="NtQueryDirectoryFile"
State  19: S=7
State  19: ProcessName="Weakas.exe"
```

State 19: PID:2132 issued a call in cat1 *

State 20: Call="ReadFile"

State 20: S=8

State 20: ProcessName="Weakas.exe"

State 20: The file ((Weakas.exe)) issued calls in the previous two categories

State 20: And a call issued in cat3 *

State 20: _____Then_____(it's a candidate virus)_____

State 22: Call="NtWriteFile"

State 22: S=9

State 22: ProcessName="Weakas.exe"

State 22: A call issued in cat4 *

State 24: Call="NtSetInformationFile"

State 24: S=10

State 24: ProcessName="Weakas.exe"

State 24: And a call issued in cat5 *

State 25: Call="ReadFile"

State 25: S=11

State 25: ProcessName="Weakas.exe"

State 25: The file ((Weakas.exe)) is a COMPUTER VIRUS *

Example 2:

In this example the viral process *Eliles.exe* will be tracked. This malware is a computer worm that is considered to be a computer virus because it attaches itself to another file prior to spread itself across the network. This malware follows the same order which *Weakas.exe* virus follows (Cat2→ Cat1→ Cat3→ Cat4→ Cat5) as shown in Figure A.1. The first snapshot (Listing A.4) shows how Tempura will read the Native and Win32 API calls issued by this virus from Deviare API. Secondly, how the final result can be achieved when the virus attempts to infect another file and follows the five steps with their rules, will be shown in the second snapshot (Listing A.5).

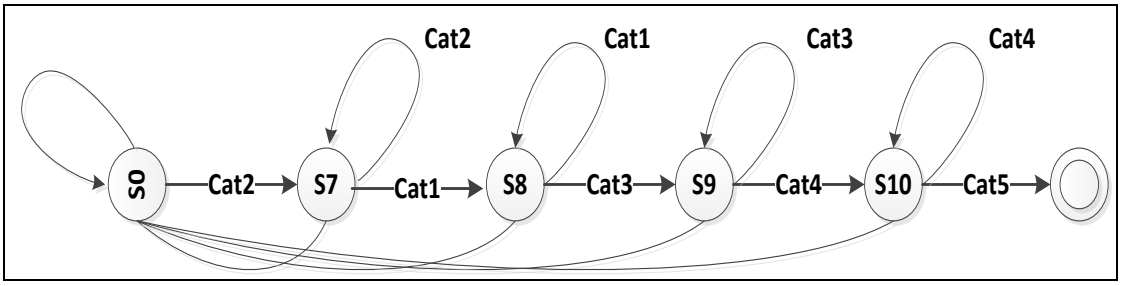


Figure A.1: The order Weakas and Eliles follow to attach themselves to another file.

Listing A.4: Snapshot of Eliles.exe Assertion.

```
!PROG: assert Name:Eliles.exe:Id:3422:Call:GetFileAttributesW:Inputs::!

!PROG: assert Name:Eliles.exe:Id:3422:Call:GetCompressedFileSizeW:Inputs::!

!PROG: assert Name:Eliles.exe:Id:3422:Call:NtQueryAttributesFile:Inputs::!

!PROG: assert Name:Eliles.exe:Id:3422:Call:NtQueryInformationFile:Inputs::!

!PROG: assert Name:Eliles.exe:Id:3422:Call:FindFirstFileW:Inputs::!

!PROG: assert Name:Eliles.exe:Id:3422:Call:FindClose:Inputs::!

!PROG: assert Name:Eliles.exe:Id:3422:Call:NtQueryDirectoryFile:Inputs::!

!PROG: assert Name:Eliles.exe:Id:3422:Call:NtQueryDirectoryFile:Inputs::!

!PROG: assert Name:Eliles.exe:Id:3422:Call:NtQueryDirectoryFile:Inputs::!

!PROG: assert Name:Eliles.exe:Id:3422:Call:ReadFile:Inputs:Eliles.exe:!

!PROG: assert Name:Eliles.exe:Id:3422:Call:NtReadFile:Inputs:Eliles.exe:!

!PROG: assert Name:Eliles.exe:Id:3422:Call:WriteFile:Inputs::!

!PROG: assert Name:Eliles.exe:Id:3422:Call:NtWriteFile:Inputs::!

!PROG: assert Name:Eliles.exe:Id:3422:Call:SetFileAttributesW:Inputs::!

!PROG: assert Name:Eliles.exe:Id:3422:Call:SetFileShortNameW:Inputs::!

!PROG: assert Name:Eliles.exe:Id:3422:Call:NtSetInformationFile:Inputs::!
```

Listing A.5: Snapshot of Eliles.exe log file (after detection).

```
State  54: Call="GetFileAttributesW"

State  54: S=0

State  54: ProcessName="Eliles.exe"

State  54:  PID:3422 issued a call in cat2                    *

State  55: Call="GetCompressedFileSizeW"

State  55: S=7

State  55: ProcessName="Eliles.exe"

State  56: Call="NtQueryInformationFile"

State  56: S=7

State  56: ProcessName="Eliles.exe"

State  57: Call="NtQueryInformationFile"

State  57: S=7

State  57: ProcessName="Eliles.exe"

State  58: Call="FindFirstFileW"

State  58: S=7

State  58: ProcessName="Eliles.exe"

State  58:  PID:3422 issued a call in cat1                    *

State  59: Call="FindClose"

State  59: S=8

State  59: ProcessName="Eliles.exe"

State  60: Call="NtQueryDirectoryFile"

State  60: S=8
```

```
State  60: ProcessName="Eliles.exe"

State  61: Call="NtQueryDirectoryFile"

State  61: S=8

State  61: ProcessName="Eliles.exe"

State  62: Call="NtQueryDirectoryFile"

State  62: S=8

State  62: ProcessName="Eliles.exe"

State  63: Call="ReadFile"

State  63: S=8

State  63: ProcessName="Eliles.exe"

State  63:  The file ( (Eliles.exe) ) issued calls in the previous two categories

State  63:  And a call issued in cat3                                    *

State  63: _____Then_______(it's a candidate virus)_____________

State  64: Call="NtReadFile"

State  64: S=9

State  64: ProcessName="Eliles.exe"

State  65: Call="WriteFileEx"

State  65: S=9

State  65: ProcessName="Eliles.exe"

State  65:  A call issued in cat4                                        *

State  66: Call="NtWriteFile"

State  66: S=10

State  66: ProcessName="Eliles.exe"
```

```
State  67: Call="SetFileAttributesW"

State  67: S=10

State  67: ProcessName="Eliles.exe"

State  67:  And a call issued in cat5                                    *

State  68: Call="SetFileShortNameW"

State  68: S=11

State  68: ProcessName="Eliles.exe"

State  68:  The file ( (Eliles.exe) ) is a COMPUTER VIRUS                *
```

Appendix B

API Intercepting Source Code

Appendix B is the C# source code that is used by Deviare API tool to intercept Win32 and Native API calls. It can be seen that only 68 calls are intercepted in this code as shown in Listing B.2. The assertion points which contain the process name, Id, the call, and the parameters associated with this call, will go through several steps each time a call is issued. Firstly, Deviare API needs to extract both calls with their parameters as assertion points in order to be read by Tempura, as shown in Listing B.1. Then, the Java pipe needs to be able to read these assertion points and then delivers them to Tempura. Finally, Tempura should be able to read and examine them.

Listing B.1: C# Assertion Points.

```
//Assertion Points
                Console.WriteLine("!PROG: assert
Name:{0}:Id:{1}:Call:{2}:Inputs:{3}:!", P_name, P_ID, Call, hfile);
```

Listing B.2: C# Source code.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.IO;
using System.Windows.Forms;
using Nektra.Deviare2;


namespace TestHook
{
    public partial class Form1 : Form
    {
        private NktSpyMgr _spyMgr;
        private NktProcess _process;
        private NktTools _tool;
        private StreamWriter _writer;


        public Form1()
        {
            int res;
            _spyMgr = new NktSpyMgr();
            _tool = new NktTools();
            res = _spyMgr.Initialize();
            _spyMgr.OnFunctionCalled += new
DNktSpyMgrEvents_OnFunctionCalledEventHandler(OnFunctionCalled);
            _writer = new
StreamWriter(@"C:\Users\Public\TestFolder\WriteText.txt");



            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Hooking Process

            //  First Category -Find to infect-

            NktHook hook3 =
_spyMgr.CreateHook("kernel32.dll!FindFirstStreamW", (int)(0));
            NktHook hook4 =
_spyMgr.CreateHook("kernel32.dll!FindFirstFileTransactedW",
(int)(0));
            NktHook hook5 =
_spyMgr.CreateHook("kernel32.dll!FindFirstStreamTransactedW",
(int)(0));
            NktHook hook6 =
_spyMgr.CreateHook("kernel32.dll!FindClose", (int)(0));
            NktHook hook7 =
_spyMgr.CreateHook("kernel32.dll!FindNextFileW", (int)(0));
```

```
            NktHook hook8 =
_spyMgr.CreateHook("kernel32.dll!FindFirstFileNameW", (int)(0));
            NktHook hook9 =
_spyMgr.CreateHook("kernel32.dll!FindNextFileNameW", (int)(0));
            NktHook hook10 =
_spyMgr.CreateHook("kernel32.dll!FindFirstFileNameTransactedW",
(int)(0));
            NktHook hook11 =
_spyMgr.CreateHook("kernel32.dll!FindNextStreamW", (int)(0));
            NktHook hook12 =
_spyMgr.CreateHook("kernel32.dll!FindFirstFileExW", (int)(0));
            NktHook hook121 =
_spyMgr.CreateHook("kernel32.dll!FindFirstFileW", (int)(0));


            //Second Category -Get information-

            NktHook hook13 =
_spyMgr.CreateHook("kernel32.dll!GetFileAttributesExW", (int)(0));
            NktHook hook14 =
_spyMgr.CreateHook("kernel32.dll!GetFileAttributesTransactedW",
(int)(0));
            NktHook hook15 =
_spyMgr.CreateHook("kernel32.dll!GetFileAttributesW", (int)(0));
            NktHook hook16 =
_spyMgr.CreateHook("kernel32.dll!GetFileInformationByHandle",
(int)(0));
            NktHook hook17 =
_spyMgr.CreateHook("kernel32.dll!GetFileBandwidthReservation",
(int)(0));
            NktHook hook18 =
_spyMgr.CreateHook("kernel32.dll!GetCompressedFileSizeTransactedW",
(int)(0));
            NktHook hook19 =
_spyMgr.CreateHook("kernel32.dll!GetFileInformationByHandleEx",
(int)(0));
            NktHook hook20 =
_spyMgr.CreateHook("kernel32.dll!GetCompressedFileSizeW", (int)(0));
            NktHook hook21 =
_spyMgr.CreateHook("kernel32.dll!GetBinaryTypeW", (int)(0));
            NktHook hook22 =
_spyMgr.CreateHook("kernel32.dll!GetFileSizeEx", (int)(0));
            NktHook hook23 =
_spyMgr.CreateHook("kernel32.dll!GetFileSize", (int)(0));
            NktHook hook24 =
_spyMgr.CreateHook("kernel32.dll!GetFileType", (int)(0));
            NktHook hook25 =
_spyMgr.CreateHook("kernel32.dll!GetTempFileNameW", (int)(0));
            NktHook hook26 =
_spyMgr.CreateHook("kernel32.dll!GetFinalPathNameByHandleW",
(int)(0));
            NktHook hook27 =
_spyMgr.CreateHook("kernel32.dll!GetLongPathNameTransactedW",
(int)(0));
            NktHook hook28 =
_spyMgr.CreateHook("kernel32.dll!GetFullPathNameTransactedW",
(int)(0));
            NktHook hook29 =
_spyMgr.CreateHook("kernel32.dll!GetFullPathNameW", (int)(0));
```

```
            NktHook hook30 =
_spyMgr.CreateHook("kernel32.dll!GetLongPathNameW", (int)(0));
            NktHook hook31 =
_spyMgr.CreateHook("kernel32.dll!GetShortPathNameW", (int)(0));


            //Third Category -Read and/or copy-

            NktHook hook2 =
_spyMgr.CreateHook("kernel32.dll!ReadFile", (int)(0));
            NktHook hook222 =
_spyMgr.CreateHook("kernel32.dll!ReadFileEx", (int)(0));
            NktHook hook =
_spyMgr.CreateHook("kernel32.dll!CreateFileW", (int)(0));
            NktHook hook32 =
_spyMgr.CreateHook("kernel32.dll!OpenFile", (int)(0));
            NktHook hook33 =
_spyMgr.CreateHook("kernel32.dll!OpenFileById", (int)(0));
            NktHook hook34 =
_spyMgr.CreateHook("kernel32.dll!ReOpenFile", (int)(0));
            NktHook hook35 =
_spyMgr.CreateHook("kernel32.dll!CreateHardLinkTransactedW",
(int)(0));
            NktHook hook36 =
_spyMgr.CreateHook("kernel32.dll!CreateHardLinkW", (int)(0));
            NktHook hook37 =
_spyMgr.CreateHook("kernel32.dll!CreateSymbolicLinkTransactedW",
(int)(0));
            NktHook hook38 =
_spyMgr.CreateHook("kernel32.dll!CreateSymbolicLinkW", (int)(0));
            NktHook hook39 =
_spyMgr.CreateHook("kernel32.dll!CopyFileExW", (int)(0));
            NktHook hook40 =
_spyMgr.CreateHook("kernel32.dll!CopyFileW", (int)(0));
            NktHook hook41 =
_spyMgr.CreateHook("kernel32.dll!CopyFileTransactedW", (int)(0));


            //Fourth Category -Write and/or delete-

            NktHook hook42 =
_spyMgr.CreateHook("kernel32.dll!ReplaceFileW", (int)(0));
            NktHook hook43 =
_spyMgr.CreateHook("kernel32.dll!WriteFile", (int)(0));
            NktHook hook434 =
_spyMgr.CreateHook("kernel32.dll!WriteFileEx", (int)(0));

            NktHook hook44 =
_spyMgr.CreateHook("kernel32.dll!DeleteFileTransactedW", (int)(0));
            NktHook hook45 =
_spyMgr.CreateHook("kernel32.dll!CloseHandle", (int)(0));
            NktHook hook46 =
_spyMgr.CreateHook("kernel32.dll!DeleteFileW", (int)(0));


            //Fifth Category -Set information-

            NktHook hook47 =
_spyMgr.CreateHook("kernel32.dll!SetFileInformationByHandle",
```

```
(int)(0));
            NktHook hook48 =
_spyMgr.CreateHook("kernel32.dll!SetFileValidData", (int)(0));
            NktHook hook49 =
_spyMgr.CreateHook("kernel32.dll!SetFileBandwidthReservation",
(int)(0));
            NktHook hook50 =
_spyMgr.CreateHook("kernel32.dll!SetFileShortNameW", (int)(0));
            NktHook hook51 =
_spyMgr.CreateHook("kernel32.dll!SetFileAttributesTransactedW",
(int)(0));
            NktHook hook52 =
_spyMgr.CreateHook("kernel32.dll!SetFileApisToOEM", (int)(0));
            NktHook hook53 =
_spyMgr.CreateHook("kernel32.dll!SetFileAttributesW", (int)(0));
            NktHook hook54 =
_spyMgr.CreateHook("kernel32.dll!SetFileApisToANSI", (int)(0));

            NktHook hook56 =
_spyMgr.CreateHook("ntdll.dll!NtQueryDirectoryFile", (int)(0));

            NktHook hook57 =
_spyMgr.CreateHook("ntdll.dll!NtQueryAttributesFile", (int)(0));
            NktHook hook58 =
_spyMgr.CreateHook("ntdll.dll!NtQueryInformationFile", (int)(0));

            NktHook hook59 =
_spyMgr.CreateHook("ntdll.dll!NtOpenFile", (int)(0));
            NktHook hook60 =
_spyMgr.CreateHook("ntdll.dll!NtReadFile", (int)(0));
            NktHook hook61 =
_spyMgr.CreateHook("ntdll.dll!NtCreateFile", (int)(0));

            NktHook hook62 =
_spyMgr.CreateHook("ntdll.dll!NtWriteFile", (int)(0));
            NktHook hook63 =
_spyMgr.CreateHook("ntdll.dll!NtDeleteFile", (int)(0));
            NktHook hook64 = _spyMgr.CreateHook("ntdll.dll!NtClose",
(int)(0));

            NktHook hook65 =
_spyMgr.CreateHook("ntdll.dll!NtSetInformationFile", (int)(0));


            INktProcessesEnum enumProcs = _spyMgr.Processes();
            foreach (INktProcess proc in enumProcs)
            {
                try
                {
                    if (proc.Name.ToLower() == "chrome.exe" ||
proc.Name.ToLower() == "firefox.exe")
                    {
                        //Attaching proces for all the categories


                        hook3.Attach(proc,true);
                        hook4.Attach(proc, true);
                        hook5.Attach(proc, true);
                        hook6.Attach(proc, true);
```

```
                    hook7.Attach(proc, true);
                    hook8.Attach(proc, true);
                    hook9.Attach(proc, true);
                    hook10.Attach(proc, true);
                    hook11.Attach(proc, true);
                    hook12.Attach(proc, true);
                    hook121.Attach(proc, true);


                    hook13.Attach(proc, true);
                    hook14.Attach(proc, true);
                    hook15.Attach(proc, true);
                    hook16.Attach(proc, true);
                    hook17.Attach(proc, true);
                    hook18.Attach(proc, true);
                    hook19.Attach(proc, true);
                    hook20.Attach(proc, true);
                    hook21.Attach(proc, true);
                    hook22.Attach(proc, true);
                    hook23.Attach(proc, true);
                    hook24.Attach(proc, true);
                    hook25.Attach(proc, true);
                    hook26.Attach(proc, true);
                    hook27.Attach(proc, true);
                    hook28.Attach(proc, true);
                    hook29.Attach(proc, true);
                    hook30.Attach(proc, true);
                    hook31.Attach(proc, true);


                    hook.Attach(proc, true);
                    hook2.Attach(proc, true);
                    hook222.Attach(proc, true);

                    hook32.Attach(proc, true);
                    hook33.Attach(proc, true);
                    hook34.Attach(proc, true);
                    hook35.Attach(proc, true);
                    hook36.Attach(proc, true);
                    hook37.Attach(proc, true);
                    hook38.Attach(proc, true);
                    hook39.Attach(proc, true);
                    hook40.Attach(proc, true);
                    hook41.Attach(proc, true);


                    hook42.Attach(proc, true);
                    hook43.Attach(proc, true);
                    hook434.Attach(proc, true);
                    hook44.Attach(proc, true);
                    hook45.Attach(proc, true);
                    hook46.Attach(proc, true);


                    hook47.Attach(proc, true);
                    hook48.Attach(proc, true);
                    hook49.Attach(proc, true);
                    hook50.Attach(proc, true);
                    hook51.Attach(proc, true);
```

```
                    hook52.Attach(proc, true);
                    hook53.Attach(proc, true);
                    hook54.Attach(proc, true);

                    hook56.Attach(proc, true);

                    hook57.Attach(proc, true);
                    hook58.Attach(proc, true);

                    hook59.Attach(proc, true);
                    hook60.Attach(proc, true);
                    hook62.Attach(proc, true);

                    hook62.Attach(proc, true);
                    hook63.Attach(proc, true);
                    hook64.Attach(proc, true);

                    hook65.Attach(proc, true);


                }
            }
            catch (Exception)
            {
            }
        }

        hook3.Hook(true);
        hook4.Hook(true);
        hook5.Hook(true);
        hook6.Hook(true);
        hook7.Hook(true);
        hook8.Hook(true);
        hook9.Hook(true);
        hook10.Hook(true);
        hook11.Hook(true);
        hook12.Hook(true);
        hook121.Hook(true);

        hook13.Hook(true);
        hook14.Hook(true);
        hook15.Hook(true);
        hook16.Hook(true);
        hook17.Hook(true);
        hook18.Hook(true);
        hook19.Hook(true);
        hook20.Hook(true);
        hook21.Hook(true);
        hook22.Hook(true);
        hook23.Hook(true);
        hook24.Hook(true);
        hook25.Hook(true);
        hook26.Hook(true);
        hook27.Hook(true);
        hook28.Hook(true);
        hook29.Hook(true);
        hook30.Hook(true);
        hook31.Hook(true);
```

```
            hook.Hook(true);
            hook2.Hook(true);
            hook222.Hook(true);
            hook32.Hook(true);
            hook33.Hook(true);
            hook34.Hook(true);
            hook35.Hook(true);
            hook36.Hook(true);
            hook37.Hook(true);
            hook38.Hook(true);
            hook39.Hook(true);
            hook40.Hook(true);
            hook41.Hook(true);


            hook42.Hook(true);
            hook43.Hook(true);
            hook434.Hook(true);
            hook44.Hook(true);
            hook45.Hook(true);
            hook46.Hook(true);

            hook47.Hook(true);
            hook48.Hook(true);
            hook49.Hook(true);
            hook50.Hook(true);
            hook51.Hook(true);
            hook52.Hook(true);
            hook53.Hook(true);
            hook54.Hook(true);

            hook56.Hook(true);

            hook57.Hook(true);
            hook58.Hook(true);

            hook59.Hook(true);
            hook60.Hook(true);
            hook61.Hook(true);

            hook62.Hook(true);
            hook63.Hook(true);
            hook64.Hook(true);

            hook65.Hook(true);

        }

        private bool GetProcess(string proccessName)
        {
            NktProcessesEnum enumProcess = _spyMgr.Processes();
            NktProcess tempProcess = enumProcess.First();
            while (tempProcess != null)
            {
                if (tempProcess.Name.Equals(proccessName,
StringComparison.InvariantCultureIgnoreCase) &&
tempProcess.PlatformBits > 0 && tempProcess.PlatformBits <=
IntPtr.Size * 8)
```

```
                {
                    _process = tempProcess;
                    return true;
                }
                tempProcess = enumProcess.Next();
            }

            _process = null;
            return false;
        }

        private void OnFunctionCalled(NktHook hook, NktProcess
process, NktHookCallInfo hookCallInfo)
        {
            if (hookCallInfo.IsPreCall)
            {
                string strCreateFile;
                string strCreateFilea;
                INktParamsEnum paramsEnum = hookCallInfo.Params();

                string function = "";
                function = hook.FunctionName;
                int r = function.LastIndexOf('!');
                function = function.Substring(r + 1);

                strCreateFilea = " \n" + process.Name + "     " +
process.Id.ToString() + "    " + function + "    ";
                strCreateFile = "";



                if (hook.FunctionName == "kernel32.dll!CreateFileW")
                {
                    /*
                     //lpFileName
                     INktParam param = paramsEnum.First();
                     strCreateFile += param.ReadString() + "\", ";


                     //dwDesiredAccess
                     param = paramsEnum.Next();
                     if ((param.LongVal & 0x80000000) == 0x80000000)
                         strCreateFile += "GENERIC_READ ";
                     else if ((param.LongVal & 0x40000000) ==
0x40000000)
                         strCreateFile += "GENERIC_WRITE ";
                     else if ((param.LongVal & 0x20000000) ==
0x20000000)
                         strCreateFile += "GENERIC_EXECUTE ";
                     else if ((param.LongVal & 0x10000000) ==
0x10000000)
                         strCreateFile += "GENERIC_ALL ";
                     else
                         strCreateFile += "0";
                     strCreateFile += ", ";

                     //dwShareMode
                     param = paramsEnum.Next();
                     if ((param.LongVal & 0x00000001) == 0x00000001)
```

```
                        strCreateFile += "FILE_SHARE_READ ";
                    else if ((param.LongVal & 0x00000002) ==
0x00000002)
                        strCreateFile += "FILE_SHARE_WRITE ";
                    else if ((param.LongVal & 0x00000004) ==
0x00000004)
                        strCreateFile += "FILE_SHARE_DELETE ";
                    else
                        strCreateFile += "0";
                    strCreateFile += ", ";

                    //lpSecurityAttributes
                    param = paramsEnum.Next();
                    if (param.PointerVal != IntPtr.Zero)
                    {
                        strCreateFile += "SECURITY_ATTRIBUTES(";

                        INktParamsEnum paramsEnumStruct =
param.Evaluate().Fields();
                        INktParam paramStruct =
paramsEnumStruct.First();

                        strCreateFile +=
paramStruct.LongVal.ToString();
                        strCreateFile += ", ";

                        paramStruct = paramsEnumStruct.Next();
                        strCreateFile +=
paramStruct.PointerVal.ToString();
                        strCreateFile += ", ";

                        paramStruct = paramsEnumStruct.Next();
                        strCreateFile +=
paramStruct.LongVal.ToString();
                        strCreateFile += ")";
                    }
                    else
                        strCreateFile += "0";
                    strCreateFile += ", ";

                    //dwCreationDisposition
                    param = paramsEnum.Next();
                    if (param.LongVal == 1)
                        strCreateFile += "CREATE_NEW ";
                    else if (param.LongVal == 2)
                        strCreateFile += "CREATE_ALWAYS ";
                    else if (param.LongVal == 3)
                        strCreateFile += "OPEN_EXISTING ";
                    else if (param.LongVal == 4)
                        strCreateFile += "OPEN_ALWAYS ";
                    else if (param.LongVal == 5)
                        strCreateFile += "TRUNCATE_EXISTING ";
                    else
                        strCreateFile += "0";
                    strCreateFile += ", ";

                    //dwFlagsAndAttributes
                    strCreateFile += param.LongVal;
                    strCreateFile += ", ";
```

```
                    //hTemplateFile
                    strCreateFile += param.LongLongVal;
                    strCreateFile += ");\r\n";
                    */
                }

                else if (hook.FunctionName == "kernel32.dll!ReadFile"
|| hook.FunctionName == "kernel32.dll!ReadFileEx"
                    || hook.FunctionName == "ntdll.dll!NtReadFile")
                {
                    IntPtr h = paramsEnum.GetAt(0).SSizeTVal;
                    string s = "";
                    if (h != IntPtr.Zero)
                    {
                        try
                        {
                            s = _tool.GetFileNameFromHandle(h,
process);
                        }
                        catch (Exception e)
                        {
                            throw (e);
                        }
                    }

                    int i = s.LastIndexOf('\\');
                    s = s.Substring(i + 1);

                    strCreateFile += s;
                    strCreateFilea += s;
                }

                Output(strCreateFilea);
                Output1(process.Name, process.Id.ToString(),
function, strCreateFile);
            }

        }

        public delegate void OutputDelegate(string P_name, string
P_ID, string Call, string hfile );
        private void Output1(string P_name, string P_ID, string Call,
string hfile)
        {
            if (InvokeRequired)
                BeginInvoke(new OutputDelegate(Output1), P_name,P_ID,
Call, hfile);
            else
            {
                    //Assertion Points
                Console.WriteLine("!PROG: assert
Name:{0}:Id:{1}:Call:{2}:Inputs:{3}:!", P_name, P_ID, Call, hfile);

            }

        }

        private void Output(string strOutput)
```

```
        {

            lock (_writer)
            {
                _writer.WriteLine(strOutput);
            }



        }

        }
    }
```

Appendix C

Tempura Source Code

Appendix C is the Tempura code which is the testing code that decides whether a process is a benign or viral process. The code shown in Listing C.1, is the one that makes it possible to read the four parameters coming from Deviare API through the Java pipe. When a call is received, Tempura will examine the four parameters to see whether the five categories are issued by a certain process or not. In addition, the orders shown in Figure 5.5 need also to be taken into account.

Listing C.1: The four parameters.

```
define get_var(X0,X1,X2,X3,Y0,Y1,Y2,Y3) =
{
   exists T : {
      get2(T) and
      if Pnamer(T)=X0 then {Y0=Pnamel(T)}
                  and
      if Pidr(T)=X1 then {Y1=Pidl(T)}
                  and
      if Callr(T)=X2 then {Y2=Calll(T)}
```

```
                and
    if Filenamer(T)=X3 then {Y3=Filenamel(T)} }
```

The following Figure shows that the “External” interface of Tempura will indicate whether both the *TestHook.exe* that intercepts Win32 and Native API calls, and the Java pipe which delivers these calls to Tempura, are successfully initialised and ready to be used by Tempura. If yes, this means that Tempura is able to receive both calls at the same time. In addition, the role of Tempura will now start and it can tell whether a process attempts to attach itself to another file or not.

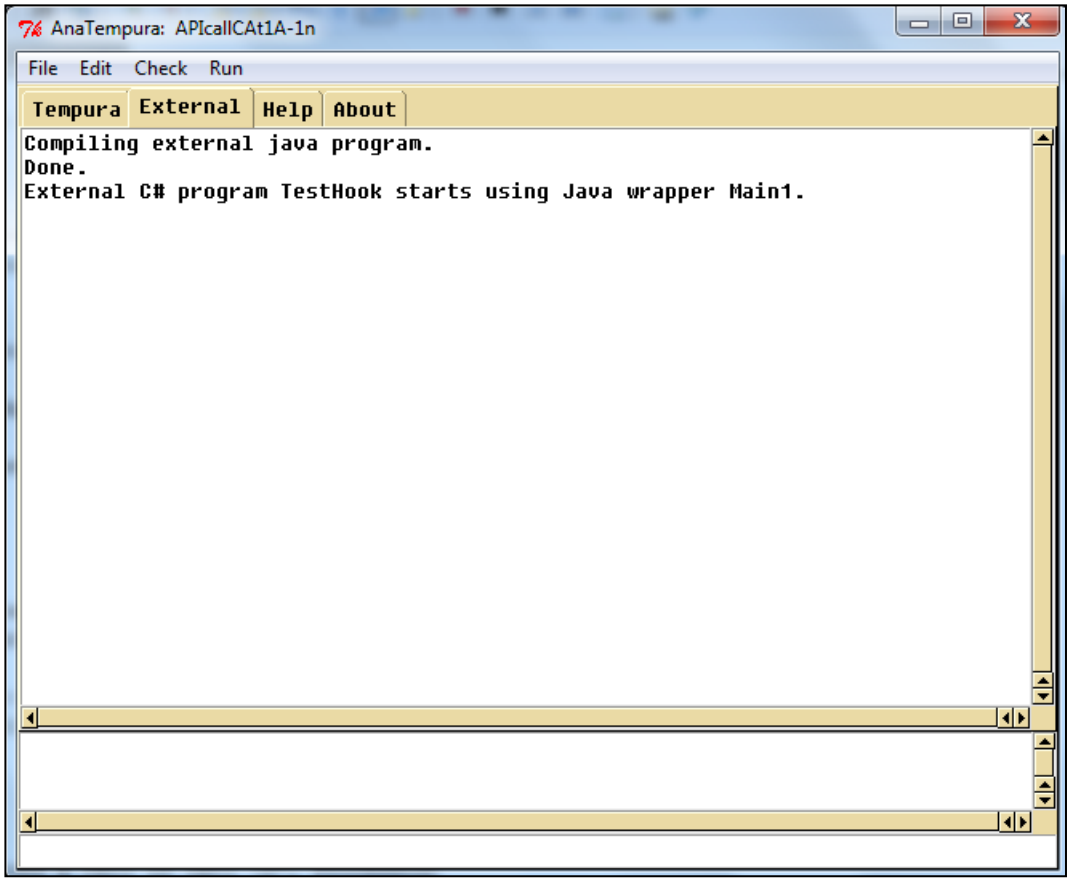


Figure C.1: Tempura Integration.

The following listing is the full Tempura code which is used in this research to examine both Win32 and Native API calls at runtime.

Listing C.2: Tempura Source code.

```
/*
 * Behavioural virus detection system Implementation
 * Sulaiman Al amro
 * 15/03/2012
 *
 *
 *
 */
/*
 * We model this system with the following Lists:
 * Cat1 : Find to infect category
 * Cat2 : Get information category
 * Cat3 : Read and/or copy category
 * Cat4 : Write and/or delete category
 and
 * Cat5 : Set Information category
 *
 *
 *
 */

 /* The following Functions has been used:
 * StartInit : To initialise every list
 * SysTestMonitoring : is the main of our program
 * CheckApiCalls : Check which API belongs to which category
 *
 *
 *
 */
```

```
/*-----------------------Find to infect category-----------------------------*/

define cat1       = ["FindFirstStreamW",
                     "FindFirstFileTransactedW",
                     "FindFirstStreamTransactedW",
                     "FindClose",
                     "FindNextFileW",
                     "FindFirstFileNameW",
                     "FindNextFileNameW",
                     "FindFirstFileNameTransactedW",
                     "FindNextStreamW",
                     "FindFirstFileExW",
                     "FindFirstFileW",
                     "NtQueryDirectoryFile"].

/*-----------------------Get information category---------------------------*/

define cat2       = ["GetFileAttributesExW",
                     "GetFileAttributesTransactedW",
                     "GetFileAttributesW",
                     "GetFileInformationByHandle",
                     "GetFileBandwidthReservation",
                     "GetCompressedFileSizeTransactedW",
                     "GetFileInformationByHandleEx",
                     "GetCompressedFileSizeW",
                     "GetBinaryTypeW",
                     "GetFileSizeEx",
                     "GetFileSize",
                     "GetFileType",
```

```
                "GetTempFileNameW",
                "GetFinalPathNameByHandleW",
                "GetLongPathNameTransactedW",
                "GetFullPathNameTransactedW",
                "GetFullPathNameW",
                "GetLongPathNameW",
                "GetShortPathNameW",
                "NtQueryAttributesFile",
                "NtQueryInformationFile"].

/*----------------------Read and/or copy category-----------------------------*/

define cat3                = ["ReadFile",
                   "ReadFileEx",
                   "OpenFile",
                   "OpenFileById",
                   "ReOpenFile",
                   "NtOpenFile",
                   "NtReadFile"].

define cat3A       = ["CreateHardLinkTransactedW",
                "CreateHardLinkW",
                "CreateSymbolicLinkTransactedW",
                "CreateSymbolicLinkW",
                "CreateFileW",
                "CopyFileExW",
                "CopyFileW",
                "CopyFileTransactedW",
                "NtCreateFile"].
```

```
/*----------------------Write and/or delete category--------------------------*/

define cat4       = ["WriteFile",
                     "WriteFileEx",
                    "ReplaceFileW",
                    "NtWriteFile"].

define cat4A       =["DeleteFileW",
                "DeleteFileTransactedW",
                "CloseHandle",
                "NtDeleteFile",
                "NtClose"].

/*---------------------------------Set Information category--------------------*/

define cat5       = ["SetFileInformationByHandle",
                "SetFileValidData",
                "SetFileBandwidthReservation",
                "SetFileShortNameW",
                "SetFileAttributesTransactedW",
                "SetFileApisToOEM",
                "SetFileAttributesW",
                "SetFileApisToANSI",
                "NtSetInformationFile"].

define extend_list(L,d) = {
   list(next L, |L|+1) and
   forall i<|L|+1: if i<|L| then L[i]:=L[i] else L[i]:=d
}.
```

```
load "../library/conversion".
load "../library/exprog".

/* csc TestHook Main1 0 */
set print_states = true.
define get_var(X0,X1,X2,X3,Y0,Y1,Y2,Y3) =
{
   exists T : {
      get2(T) and
      if Pnamer(T)=X0 then {Y0=Pnamel(T)}
                  and
      if Pidr(T)=X1 then {Y1=Pidl(T)}
                  and
      if Callr(T)=X2 then {Y2=Calll(T)}
                  and
      if Filenamer(T)=X3 then {Y3=Filenamel(T)}

   }
}.
/*-------------------------------------------------------------------------*/
/* run */ define SysTestMonitoring() = {


 exists  Name, Id, Na, Sa, Inputs, I, H, K,: {

/*-----------------------CheckApiCalls---------------------------------------*/

 define    CheckApiCalls(N,S,ProcessName, ProcessId, Call,Inputs)=
```

```
exists  J : {
                    skip and output(Call) and output(S) and
                    output(ProcessName) and
          if N ~= ProcessName then { S:=S }
          else {
          if S=0 and (exists i<|cat1|:
                                        cat1[i]=Call)
          then {
                                        S:=1
and
          format(" PID:%s issued a call in cat1 \n", ProcessId)
    }
and
          if S=0 and (exists i<|cat2|:
                                        cat2[i]=Call)
            then {
                                         S:=7
and
              format(" PID:%s issued a call in cat2 \n", ProcessId)
                }
and

           if S=0 and not( (exists i<|cat1|: cat1[i]=Call) or (exists i<|cat2|: cat2[i]=Call))
           then {
                                        S:= 0
           }
 and
          if S=1 and (exists i<|cat1|:
                                              cat1[i]=Call)
          then {
```

```
                                    S:=1
          }
and
          if S=1 and (exists i<|cat2|:
                                    cat2[i]=Call)
          then {
                                    S:=2
and
            format(" PID:%s issued a call in cat2 \n", ProcessId)
            }
and
           if S=1 and not( (exists i<|cat1|: cat1[i]=Call) or (exists i<|cat2|: cat2[i]=Call))
           then {


                                    S:= 0
          }
          and
          if S=2 and (exists i<|cat2|:
                                    cat2[i]=Call)
          then {
                                     S:=2
     }
   and
          if S=2 and (exists i<|cat3|:
                                          cat3[i]=Call)
          then {
                          if (ProcessName=Inputs)
                          then {
                                     S:=3
and
```

```
        format(" The file ( (%s) ) issued calls in the previous two categories \n",
ProcessName)
and
        format(" And a call issued in cat3 \n")
and
        format("_____Then_______(it's a candidate virus)__________ \n\n")


         }
         else {
                                S:= 2
        }
        }
and
         if S=2 and not( (exists i<|cat2|: cat2[i]=Call) or (exists i<|cat3|: cat3[i]=Call))
         then {
                                S:= 0
         }
and
        if S=3 and (exists i<|cat3|:
                                    cat3[i]=Call)
        then {
                                 S:=3
              }
and
        if S=3 and (exists i<|cat4|:
                                    cat4[i]=Call)
        then {
                                  S:=4
and
         format(" A call issued in cat4 \n")
```

```
                    }
and
          if S=4 and (exists i<|cat4|:
                                   cat4[i]=Call)
           then {
                                    S:=4
                    }
and
          if S=4 and (exists i<|cat5|:
                                         cat5[i]=Call)
          then {
                                    S:=5
and
           format(" And a call issued in cat5 \n")
                              }
 and
           if S=4 and not( (exists i<|cat4|: cat4[i]=Call) or (exists i<|cat5|: cat5[i]=Call))
           then {
                                    S:= 0
           }
and
          if S=3 and (exists i<|cat5|:
                                         cat5[i]=Call)
               then {
                                    S:=6
 and
           format(" A call issued in cat5 \n")
            }
 and
          if S=3 and not((exists i<|cat3|: cat3[i]=Call) or (exists i<|cat4|: cat4[i]=Call)
```

```
or (exists i<|cat5|: cat5[i]=Call))
        then {
                                        S:= 0
        }
and
        if S=6 and (exists i<|cat5|:
                                          cat5[i]=Call)
        then {
                                        S:=6
        }
 and
        if S=6 and (exists i<|cat4|:
                                          cat4[i]=Call)
        then {
                                        S:=5
 and
        format(" And a call issued in cat4 \n")
                }
 and
        if S=6 and not( (exists i<|cat4|: cat4[i]=Call) or (exists i<|cat5|: cat5[i]=Call))
        then {
                                        S:= 0
        }
and
        if (S=5)
        then {
                                        S:=5
and
        format(" The file ( (%s) ) is a COMPUTER VIRUS \n", ProcessName)
        }
```

```
and
        if S=7 and (exists i<|cat2|:
                                        cat2[i]=Call)
        then {
                                        S:=7
                }
and
        if S=7 and (exists i<|cat1|:
                                        cat1[i]=Call)
        then {
                                        S:=8
 and
         format(" PID:%s issued a call in cat2 \n", ProcessId)
                }
and
         if S=7 and not( (exists i<|cat1|: cat1[i]=Call) or (exists i<|cat2|: cat2[i]=Call))
         then {
                                         S:= 0
         }
and
        if S=8 and (exists i<|cat1|:
                                        cat1[i]=Call)
                        then {

                                         S:=8
            }
 and
        if S=8 and (exists i<|cat3|:
                                        cat3[i]=Call)
        then {
```

```
          if (ProcessName=Inputs)
          then {
                                        S:=9
 and
          format(" The file ( (%s) ) issued calls in the previous two categories \n",
ProcessName)
and
          format(" And a call issued in cat3 \n")
and
          format("_____Then_______(its a candiate virus)_____________ \n\n")
          }
          else {
                                        S:=8
          }
                    }
and
          if S=8 and not( (exists i<|cat1|: cat1[i]=Call) or (exists i<|cat3|: cat3[i]=Call))
          then {
                                        S:= 0
          }
and
          if S=9 and (exists i<|cat3|:
                                       cat3[i]=Call)
          then {
                                        S:=9
                              }
and

          if S=9 and (exists i<|cat4|:
```

```
                                        cat4[i]=Call)
        then {
                                        S:=10
 and
        format(" A call issued in cat4 \n")
                                }
and
        if S=10 and (exists i<|cat4|:
                                        cat4[i]=Call)
        then {
                                        S:=10
                }
and
        if S=10 and (exists i<|cat5|:
                                        cat5[i]=Call)
        then {
                                        S:=11
 and
         format(" And a call issued in cat5 \n")
        }
and
        if S=10 and not( (exists i<|cat4|: cat4[i]=Call) or (exists i<|cat5|: cat5[i]=Call))
         then {
                                        S:= 0
         }
and
        if S=9 and (exists i<|cat5|:
                                        cat5[i]=Call)
                then {
                                        S:=12
```

```
and
        format(" A call issued in cat5 \n")
        }
 and
         if S=9 and not((exists i<|cat3|: cat3[i]=Call) or (exists i<|cat4|: cat4[i]=Call)
or (exists i<|cat5|: cat5[i]=Call))
         then {
                                  S:= 0
         }
 and
        if S=12 and (exists i<|cat5|:
                                  cat5[i]=Call)
        then {
                                  S:=12
                }
and
        if S=12 and (exists i<|cat4|:
                                  cat4[i]=Call)
        then {
                                  S:=11
and
         format(" And a call issued in cat4 \n")
        }
 and
        if S=12 and not( (exists i<|cat4|: cat4[i]=Call) or (exists i<|cat5|: cat5[i]=Call))
         then {
                                  S:= 0
         }
and
        if (S=11)
```

```
          then {
                                        S:=11
and
       format(" The file ( (%s) ) is a COMPUTER VIRUS \n", ProcessName)
          }
                    }
                    }
}
and
/*--&&&&&&&&..... run of the System.....&&&&&&----------*/
 H=[] and K=0 and I =0 and
          {
 {
          I gets I+1} and always {if I=3000 then Done=true else Done=false } and
len(3000)
and
     list(Sa,2) and stable(struct(Sa)) and forall i<|Sa|: {Sa[i]=i} and output(Sa) and
     list(Na,2) and stable(struct(Na)) and Na[0] = "chrome.exe" and stable(Na[0]) and
Na[1] = "firefox.exe" and stable(Na[1]) and


      while(not Done) do {
            {
       {
                    get_var("Name","Id","Call","Inputs", Name,Id,Call,Inputs)
and
                     stable{Name} and
                     stable{Call} and
                     stable{Id}   and
                     stable{Inputs} and
                     (forall i< |Na| : {CheckApiCalls(Na[i], Sa[i], Name,Id,Call, Inputs)})
```

```
                    };{skip and (forall i<|Na| : {Na[i]:=Na[i] and Sa[i]:=Sa[i]})}
                    }
}
}
 {
 empty and PrintCatandApiFile()
 }
}
}.
set print_states=true.
```