

## Table of Contents

CHAPTER 1 .....	2
INTRODUCTION .....	2
1.1    PROBLEM DEFINITION AND OBJECTIVES .....	3
1.1.1 Problem definition.....	3
1.1.2 Objectives and Benefits .....	3
1.2    REPORT CONTENT SUMMARY .....	4
CHAPTER 2 .....	5
LITERATURE SURVEY .....	5
2.1 AI TECHNIQUES USED IN BEHAVIOUR BASED DETECTION .....	7
2.2 LEARNING TECHNIQUES USED IN MALWARE DETECTION.....	10
2.3 MONITORING OF MALWARE BEHAVIOUR.....	11
2.3.1 Malware Sandboxes .....	11
2.3.2 Malware Instruction Set .....	12
2.3.3 Malware Embedding Function.....	13
2.4 SUPPORT VECTOR MACHINES .....	18
2.5 EVALUATION METRICS .....	20
CHAPTER 3 .....	24
WORK DONE .....	24
3.1 DATA GATHERING PHASE.....	24
3.2 DESIGN OF SIGNATURE MATCHING TECHNIQUES .....	25
3.3 IMPLEMENTATION OF SIGNATURE BASED DETECTION .....	26
3.4 DEPENDENCY BASED CLASSIFICATION OF VIRUSES .....	27
3.5 BEHAVIOUR BASED CLASSIFICATION OF VIRUSES .....	30
3.5.1 Parameters identified for heuristic scanning .....	35
3.6    IMPLEMENTATION OF HEURISTIC SCANNING .....	36
3.6.1 Implementation details of static heuristic scanning of DOS viruses.....	36
3.6.2 Obfuscation Techniques.....	38
3.6.3 Obfuscation Handling .....	41
3.6.4 Dynamic Heuristic Scanning .....	43
3.7    MACHINE LEARNING.....	46
3.7.1    Schematic overview of framework .....	47

3.7.2	Clustering and Classification .....	49
3.7.2.1	Prototype Extraction .....	49
3.7.2.2	Clustering using Prototypes .....	50
3.7.2.3	Classification using Prototypes .....	51
3.7.3	Incremental Analysis.....	53
3.8	EVALUATION.....	54
3.8.1	Evaluation Data.....	54
3.8.2	Performance measure .....	55
CHAPTER 4	.....	56
RESULTS AND ANALYSIS	.....	56
4.1	ANALYSIS OF FRAMEWORK.....	56
4.2	F-MEASURE ANALYSIS OF LEARNING.....	58
CHAPTER 5	.....	60
SCREENSHOTS	.....	60
CHAPTER 6	.....	64
CONCLUSION AND FUTURE WORK	.....	64
6.1. WORK PLAN	.....	65
REFERENCES	.....	66
APPENDIX I	.....	70
REC STUDIO	.....	70
SUNBELT CWSANDBOX	.....	71

## Figures List

Figure No.	Page No.
Figure 1.1: Sequence flow of Anti-virus application	<b>3</b>
Figure 3.1: Flowchart for API function calls	<b>11</b>
Figure 3.2: Dependency based classification of viruses	<b>12</b>
Figure 3.3: Classification of viruses based on behaviour	<b>15</b>
Figure 3.4: Classification of Heuristic Scanning based on approach	<b>21</b>
Figure 3.5: Machine Language and Assembly code for terminating a program – permutation 1	<b>22</b>
Figure 3.6: Machine Language and Assembly code for terminating a program – permutation 2	<b>22</b>
Figure 3.7: Flowchart for static heuristic scanning	<b>23</b>
Figure 4.1: Results and Analysis	<b>24</b>
Figure 5.1: Screenshot 1 of signature based detection	<b>59</b>
Figure 5.2: Screenshot 2 of signature based detection	<b>59</b>
Figure 5.3: Screenshot of static heuristic scanning	<b>60</b>
Figure 5.4: Screenshot of decompilation	<b>60</b>
Figure 5.5: Screenshot of hexadecimal dump	<b>61</b>
Figure 5.6: Screenshot of dynamic heuristic scanning	<b>62</b>
Figure 5.7: Screenshot of incremental analysis	<b>62</b>

# CHAPTER 1

## INTRODUCTION

**Malware**, short for **malicious software**, is software used or created by hackers to disrupt computer operation, gather sensitive information, or gain access to private computer systems. It mostly appears in the form of scripts or code snippets. It is a general term used to refer to a variety of forms of hostile software. Malware includes computer viruses, worms, trojans, spyware, adware, and other malicious programs. There are 3 important characteristics that are required to classify a program as a virus:

- Self-replicating— create copies of itself and spread
- Self-constructing – create a new malware by combining actions of existing ones
- Self-evolving – create mutations of itself.

The way in which these characteristics are implemented depends on individual virus. These 3 features help a virus in consuming computation power of the system, modifying code to access/modify data available, modifying code to change the functioning of a system and/or gaining access through backdoor entry into systems with critical information thereby making them malicious to computational systems.

Malware has demanded the need for anti-virus technology. With the increasing advent of malware attacks, traditional computer virus detection technology has been unable to effectively defend against viruses, especially detection and prevention of unknown viruses or mutations of known viruses. Clearly, the evolution of malware demands the need for the system to be intelligent and capable of detecting malware behavior and activity. Hence artificial intelligence is integrated with anti-virus technology in order to improve the detection of unknown viruses or mutations of known viruses.

## 1.1 PROBLEM DEFINITION AND OBJECTIVES

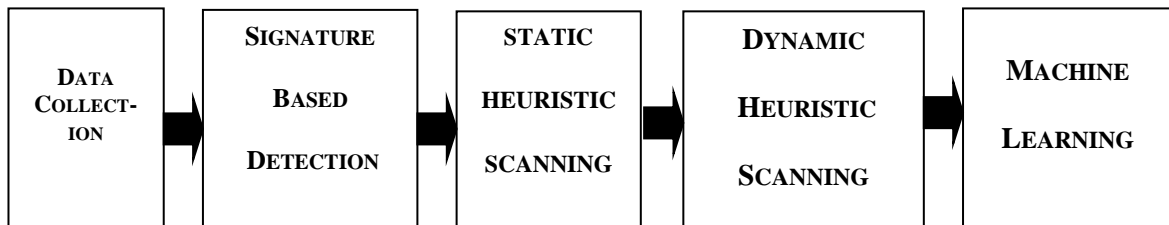
### 1.1.1 Problem definition

With the rapid development of Internet, numerous computer viruses having unimaginable capacity of attacking secure data are also increasing. Hence, there is a need for an anti-virus technology which has the capability to prevent these harmful software from attacking systems. Since antivirus technology applies to huge data sets, any mechanism should not only be reliable but also fast.

Every virus has a signature of its own. However signature detection alone cannot detect new mutations of viruses. In order to overcome this, behavior based detection or heuristic scanning is used. This technique can detect new viruses but results in false positives. Reduction of false positives is a major issue in most behavior based detection techniques. One of the most powerful ways of overcoming this is the use of machine learning techniques.

### 1.1.2 Objectives and Benefits

The objective of the framework is to achieve pro-active protection while reducing false positives



**Figure 1.1 Sequence flow of Anti-virus application**

The detection of viruses using AI can be divided into the following 5 phases –

1. Data Collection

Huge sets of data are collected in this phase. This includes all types of files out of which the definition files of the viruses need to be identified.

2. Signature Based Detection

This phase involves matching signatures of the data collected with signatures of the viruses stored in a virus gene library.

3. Static Heuristic Scanning

This phase involves static analysis of code to identify malicious code snippets/scripts embedded other executable code.

4. Dynamic Heuristic Scanning

This phase involves emulation of malicious code and monitoring of behavior ( system calls, mutex logs, call trace etc).

5. Machine Learning

This is intended to reduce false positives generated by the heuristic scanning phases while achieving pro-active protection

## **1.2 REPORT CONTENT SUMMARY**

This report is divided as follows. Chapter 2 gives a brief summary of the literature survey conducted on the various artificial intelligence techniques that can be used to combat viruses. Chapter 3 explains the work done on each of the phases of our solution model. It includes the design of the solution as well as the implementation details of the phases. Chapter 4 presents the results obtained and the analysis from our study and implementation. Chapter 5 gives the screenshots of the implemented phases. Chapter 6 explains the conclusion and work plan. This is followed by the references and an appendix.

## CHAPTER 2

### LITERATURE SURVEY

Numerous artificial intelligence techniques are being utilized to defend the system from malware attacks. Each one of them has inherent advantages and disadvantages. Each of these artificial intelligence techniques targets a specific characteristic or behaviour of the malware. We have carried out a survey of the behaviour traits of different kind of malware as well as the best artificial intelligence techniques to be employed to resist malware attacks.

The process of malware detection is known to consist of 4 steps (Xia-bin *et al.*,2008):

- Virus Detection
- Virus Elimination
- Virus Prevention
- Virus Immunity.

Virus Detection is a process that takes the suspicious file(including the boot content, the memory content and others content) as its input, perform specific detecting arithmetic, and finally output the detection result that shows whether it includes virus or not[2].There are two types of virus detection technology(Chamorro *et al.*,2012)

- Signature based detection
- Behaviour based detection.

Viruses, the most commonly found malware are not completely invisible as the malicious code needs to be stored in the memory to execute. Viruses also have specific ways of replicating and spreading themselves. All these characteristics of viruses give rise to a pattern, called a signature which is used to identify it. The scanning searches memory and monitors execution to match the expected signature of viruses.

Behaviour based detection attempts to detect virus based on indications of viral behaviour or any malware attack. Numerous artificial intelligence techniques are used to identify viruses based on behaviour which will be listed further.

Virus elimination eliminates virus from the program with virus to make it a benign program that can execute normally. Strictly, virus elimination is an extension of virus detection, because it only can eliminate virus after being detected.

Virus Prevention is that it will block the invasion or give warning while virus has not yet invasion or at the right time it starts to invade. Virus Prevention includes both, prevention of known and unknown virus. Virus prevention technology includes disk boot prevention, encrypting the executable program, read-write control technology, system monitor technology etc.

Virus immune derives from Biological Immune Technology (BIT). It can defend against all viruses in independent of virus database update.



## 2.1 AI TECHNIQUES USED IN BEHAVIOUR BASED DETECTION

The main research field of Artificial Intelligence, such as Heuristic Technology, Artificial Neural Networks etc, has become the main technique of anti-virus system. With the development of Artificial Intelligence Technology, more advanced methods, such as Data Mining Technology, Artificial Immune Technology etc, have been applied in the new generation of anti-virus detection system and play a crucial role in improving the anti-virus software's performance.

1. Heuristic Technology (Xian *et al.*, 2005, Zenhai *et al.*, 2006): Heuristic means "the ability of self-discovery", and intelligently analyse codes to detect the unknown virus by some rules while scanning (Xian *et al.*, 2005). Implementation involves a lot of complexity because it should identify and detect many suspicious code instruction sequences, such as format diskette operation, searching and locating various operations of executable programs, exit to DOS prompt, implementing resident memory operations and discovering unusual system function calls.
2. Data Mining (Matthew *et al.*, 2001, Yufeng *et al.*, 2005, Robert *et al.*, 2007): With the rapid development of Information Technology, the rapid growth of data has exceeded the ability of the manual processing of data. Extracting general knowledge from the mass of data is done using Data Mining. Data mining analyses the observed sets to discover the unknown relation and sum up the results of data analysis to make the owner of data to understand. Data Mining involves statistics, pattern identification, and machine learning, and so on. Data Mining can be used to analyse patterns of malware behaviour to help identify mutations or new malwares using their statistical and patterns.
3. Agent Technique (Lee J.S. *et al.*, 1997, T. Okamoto *et al.*, 1999, Maoguang *et al.*, 2002): Intelligent anti-virus system always is the goal of anti-virus field. With the appearance of the new unknown malicious virus, new anti-virus

techniques are generated very slowly. Thus new unknown malicious virus spread quickly without being combated. So there is growing for automatic and efficient response to such viruses. The appearance of agent in the field of Artificial Intelligence resolves this issue satisfactorily. Among these agents, antibody agents use the information of “self” (files of host computer) rather than the information of “non-self” (computer viruses). After detection and neutralization the anti-virus system tries to recover original files.

4. Artificial Immune technique (Harmer P. *et al.*, 2002, Kephart, J.O. *et al.* 1995, Marmelstein R.E. *et al.*, 1998). The traditional anti-virus techniques cannot detect successfully the new unknown viruses. Hence, it is important to seek a new method of virus detection that can be fast, accurate, and effective for detection. Artificial immune is an effective technique that derives from the principle of Biological immune system that can withstand and destroy Biological Virus even if unknown. The algorithm combines the recognition ability of immune non-self selection for unknown viruses and the memory ability of immune associative memory for known viruses.
5. Neural Networks (William *et al.*, 2000, Fukumi M. *et al.*, 2000, Chen Guo *et al.*, 2005, Boyun Zhang *et al.*, 2007, White R *et al.*, 1998): The popular methods of virus detection cannot automatically extract virus signatures. Also virus detection systems lack associative memory and the capacity of real-time calculation. They cannot automatically detect and learn, and cannot engage in large-scale parallel processing. Artificial neural networks solve this problem by taking into consideration many factors including conditionality, imprecision and fuzziness
6. Control Flow Graphs for static analysis of malware binaries - A control flow graph (CFG) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution. In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent

jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

The CFG is essential to many compiler optimizations and static analysis tools.

## **2.2 LEARNING TECHNIQUES USED IN MALWARE DETECTION**

Malicious software, referred to as malware, is one of the major threats on the Internet today. A plethora of malicious tools, ranging from classic computer viruses to Internet worms and bot networks, targets computer systems linked to the Internet. Proliferation of this threat is driven by a criminal industry which systematically comprises networked hosts for illegal purposes, such as distribution of spam messages or gathering of confidential data (Franklin et al., 2007; Holz et al., 2009). Unfortunately, the increasing amount and diversity of malware render classic security techniques, such as anti-virus scanners, ineffective and, as a consequence, millions of hosts in the Internet are currently infected with malicious software (Microsoft, 2009; Symantec, 2009).

To protect from the rapid propagation of malware in the Internet, developers of antimalware software heavily rely on the automatic analysis of novel variants for designing corresponding defense measures. The automatic analysis of malware, however, is far from a trivial task, as malware writers frequently employ obfuscation techniques, such as binary packers, encryption, or self-modifying code, to obstruct analysis. These techniques are especially effective against static analysis of malicious binaries (Linn and Debray, 2003; Christodorescu and Jha, 2003; Kruegel et al., 2005; Moser et al., 2007a; Preda et al., 2008).

In contrast to static techniques, dynamic analysis of binaries during run-time enables monitoring the behaviour of malware, which is difficult to conceal and often indicative for malicious activity. Hence, a substantial amount of research has focused on development of tools for collection and monitoring of malware (Pouget et al., 2005; Leita et al., 2006; Bacher et al., 2006; Bayer et al., 2006a,b; Willems et al., 2007; Lanzi et al., 2009).

While monitoring binaries during run-time provides means for studying the behaviour of malicious software, it is by itself not sufficient to alleviate the threat of malware proliferation. What is needed is the ability to automatically analyse the behaviour of malware binaries, such that novel strains of development can be efficiently identified and mitigated.

Two concepts for such automatic analysis of behaviour based on machine learning techniques have been recently proposed: (a) clustering of behaviour, which aims at discovering novel classes of malware with similar behaviour (Bailey et al., 2007; Bayer et al., 2009a) and (b) classification of behaviour, which enables assigning unknown malware to known classes of behaviour (Lee and Mody, 2006; Rieck et al., 2008). Mostly these concepts have been studied as competing paradigms, where either one of the two has been applied using different algorithms and representations of behaviour.

## **2.3 MONITORING OF MALWARE BEHAVIOUR**

A prerequisite for behaviour-based analysis is the efficient monitoring of malware behaviour as well as a representation of this monitored behaviour suitable for accurate analysis. In this section, we present the sandbox technique employed in our framework and describe the underlying representation of behaviour denoted as malware instruction set.

### **2.3.1 Malware Sandboxes**

For monitoring the behavior of executable binaries, multiple different methods exist from which the majority is based on the interception of system calls (Bayer et al., 2006a,b; Willems et al., 2007; Dinaburg et al., 2008).

In contrast to code analysis, where the binary to be analysed is disassembled or debugged, the actual code of the file is completely ignored under behavior-based analysis. Instead, the binary is seen as a black box and executed in a controlled environment. This environment is set up in a way, in which all system interaction of the malware is intercepted.

By detouring system calls, the sandbox can inspect—and optionally modify—all input parameters and return values of system calls during run-time of the malware binary (Hunt and Brubaker, 1999).

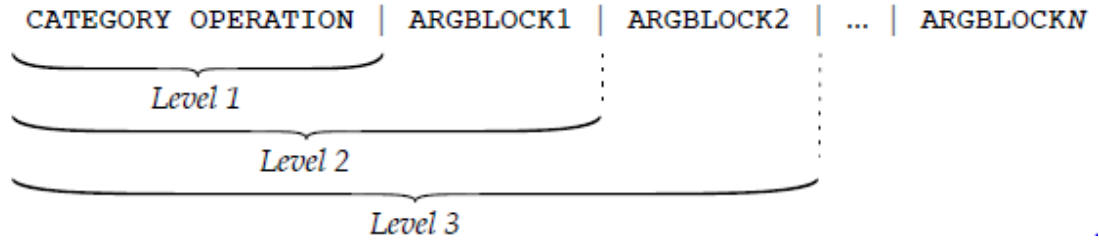
This interception can be realized on different levels, ranging from a bird's eye view in out-of-system hypervisor monitoring down to in-process monitoring realized via dynamic code instrumentation or static patching. For our analysis, we employ the

monitoring tool CWSandbox which intercepts system calls via inline function hooking. The tool overwrites the prologue of each system call with an unconditional jump to a hook function. This function first writes the system call and its arguments to a log file and then proceeds to execute the intercepted operation with all this detouring transparent to the caller.

### **2.3.2 Malware Instruction Set**

The predominant format for representation of monitored behavior are textual and XML based reports, for example, as generated by the malware sandboxes Anubis (Bayer et al., 2006b) and CWSandbox (Willems et al., 2007). While such formats are suitable for a human analyst or computation of general statistics, they are inappropriate for automatic analysis of malware behavior. The structured and often aggregated textual reports hinder application of machine learning methods, as the true sequences of observed behavioral patterns are not directly accessible. Moreover, the complexity of textual representations increases the size of reports and thus negatively impacts run-time of analysis algorithms.

To address this problem and optimize processing of reports, we propose a special representation of behavior denoted as malware instruction set (MIST) inspired from instruction sets used in processor design. In contrast to regular formats, the monitored behavior of a malware binary is described as a sequence of instructions, where individual execution flows of threads and processes are sequentially appended to a single report. Each instruction in this format encodes one monitored system call and its arguments using short numeric identifiers, such as ‘03 05’ for the system call ‘move file’. The system call arguments are arranged in blocks at different levels, reflecting behavior with different degree of specificity. We denote these levels as MIST levels. Moreover, variable-length arguments, such as file and mutex names, are represented by index numbers, where a global mapping table is used to translate between the original contents and the index numbers.



**Figure 2.1 Schematic overview of MIST instruction**

Figure 2.1 shows the basic structure of a MIST instruction. The first level of the instructions corresponds to the category and name of a monitored system call. As an example, ‘03 05’ corresponds to the category ‘filesystem’ (03) and the system call ‘move file’ (05). The following levels of the instruction contain different blocks of arguments, where the specificity of the blocks increases from left to right. The main idea underlying this rearrangement is to move “noisy” elements, such as process and thread identifiers, to the end of an instruction, whereas stable and discriminative patterns, such as directory and mutex names, are kept at the beginning. Thus, the granularity of behavior-based analysis can be adapted by considering instructions only up to a certain level. As a result, malware sharing similar behavior may be even discovered if minor parts of the instructions differ, for instance, if randomized file names are used.

### 2.3.3 Malware Embedding Function

The proposed feature representation enables an expressive characterization of behavior, where each execution of a binary is represented as a sequential report of MIST instructions. Typical behavioural patterns of malware, such as changing registry keys or modifying system files, are reflected in particular subsequences in these reports. Yet, this representation is still not suitable for application of efficient analysis techniques, as these usually operate on vectors of real numbers. To tackle this issue, we introduce a technique for embedding behavior reports in a vector space which is inspired by concepts from natural language processing and host-based intrusion detection (Salton et al., 1975; Damashek, 1995; Forrest et al., 1996; Lee et al., 1997).

### 2.3.3.1 Embedding using Instruction q-grams

In an abstract view, a report  $x$  of malware behavior corresponds to a simple sequence of instructions. To characterize the contents of this sequence, we move a fixed-length window over the report, where we consider a subsequence of length  $q$  at each position. The resulting “snippets” of instructions, referred to as instruction q-grams, reflect short behavioural patterns and thus implicitly capture some of the underlying program semantic. For constructing an embedding of reports using instruction q-grams, we consider the set  $S$  of all possible q-grams, defined as follows

$$S = \{(a_1, \dots, a_q) \mid a_i \in \mathcal{A} \text{ with } 1 \leq i \leq q\},$$

Where  $\mathcal{A}$  denotes the set of all possible instructions. Note that depending on the considered MIST level, the granularity of  $\mathcal{A}$  and  $S$  may range from plain system calls (level = 1) to full instructions covering different blocks of system call arguments (level > 1).

Using the set  $S$ , a report  $x$  of malware behavior can be embedded in an  $|S|$ -dimensional vector space, where each dimension is associated with one instruction q-gram and thus a short behavioral pattern. The corresponding embedding function  $\varphi$  resembles an indicator for the presence of instruction q-grams and can be formally defined as follows

$$\varphi(x) = (\varphi_s(x))_{s \in S} \text{ with } \varphi_s(x) = \begin{cases} 1 & \text{if report } x \text{ contains } q\text{-grams } s, \\ 0 & \text{otherwise.} \end{cases}$$

As an example, let us consider the artificial report  $x = '1|A \ 2|A \ 1|A \ 2|A'$  containing only two simplified instructions  $\mathcal{A} = \{1|A, 2|A\}$ . If we consider instruction q-grams with  $q = 2$  for characterizing the contents of  $x$ , the vector  $\varphi(x)$  looks as follows



$$\varphi('1|A \ 2|A \ 1|A \ 2|A') \longmapsto \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \begin{matrix} '1|A \ 1|A' \\ '1|A \ 2|A' \\ '2|A \ 1|A' \\ '2|A \ 2|A' \end{matrix}.$$

In contrast to this simple example, the vector space induced by real instructions exhibits a huge dimension. For example, for 2-grams with MIST level 2, a set of 1,000 reports easily exceeds over 100, 000 unique q-grams and hence is embedded in a vector space with over 100, 000 dimensions. At the first glance, computing and comparing vectors in such high-dimensional spaces seems infeasible. The number of instruction q-grams contained in a single report, however, is linear in its length. That is, a report  $x$  containing  $m$  instructions comprises at most  $(m-q)$  different q-grams.

Consequently, only  $(m-q)$  dimensions are non-zero in the feature vector  $\varphi(x)$ —irrespective of the actual dimension of the vector space. This sparsity of  $\varphi(x)$  can be exploited to derive linear-time methods for extraction and comparison of embedded reports, which ultimately enables efficient analysis of behavior as demonstrated in Section 3. A detailed discussion of linear-time methods for analysis of embedded sequences is provided by Rieck and Laskov (2008).

The number of non-zero dimensions in the vector space also depends on other factors, such as the redundancy of behavior, the considered alphabet, or the length of reports. In practice, the length of reports dominates these factors and introduces an implicit bias, rendering comparison of small and large reports problematic. To compensate this bias, we introduce a normalized embedding function

$$\hat{\varphi}(x) = \frac{\varphi(x)}{||\varphi(x)||}$$

that scales each vector  $j(x)$  such that its vector norm equals one. As a result of this normalization, a q-gram counts more in a report that has fewer distinct q-grams. That is, changing a constant amount of instructions in a report containing repetitive

behavior has more impact on the embedded vector than in a report comprising several different behavioural patterns. This type of normalization is widely used in the domain of information retrieval for comparing text documents, where it is usually applied as part of the cosine similarity measure (van Rijsbergen, 1979).

### 2.3.3.2 Comparing embedded reports

The embedding of reports in vector spaces enables expressing the similarity of behavior geometrically, which allows for designing intuitive yet powerful analysis techniques. To assess the geometric relations between embedded reports, we define a distance  $d$  by

$$d(x, z) = ||\hat{\phi}(x) - \hat{\phi}(z)|| = \sqrt{\sum_{s \in \mathcal{S}} (\hat{\phi}_s(x) - \hat{\phi}_s(z))^2}$$

which compares the behavior of the embedded reports  $x$  and  $z$ , and corresponds to the Euclidean distance in  $\mathbf{R}^{|\mathcal{S}|}$ . The values of  $d$  range from  $d(x, z) = 0$  for identical behavior to  $d(x, z) = \text{sqrt}(2)$  for maximally deviating reports due to the normalization. Access to the geometry of the induced vector space enables grouping and discriminating embedded reports effectively by means of machine learning. Malware variants originating from the same class share several instruction q-grams in their behavior and thus lie close to each other, whereas reports from different families yield large distances and are scattered in the vector space. In comparison to related approaches using distances (Lee and Mody, 2006; Bailey et al., 2007), the proposed embedding gives rise to an explicit vector representation, where the contribution of each q-gram can be traced back to individual behavioral patterns for explaining the decisions made by analysis methods

### 2.3.3.2 Clustering and Classification

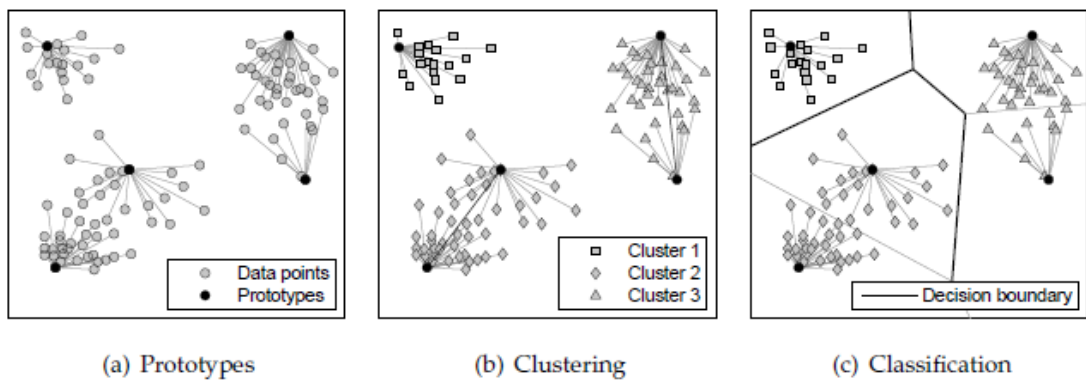
Based on the embedding of reports in a vector space, we apply techniques of machine learning for the analysis of behavior. In particular, we study two learning concepts for analysis: Clustering of behavior, which enables identifying novel classes of malware with similar behavior and classification of behavior, which allows to assign malware to known classes of behavior.

To keep abreast of the increasing amount of malware in the wild, clustering and classification methods are required to process thousands of reports on a daily basis. Unfortunately, most learning methods scale super-linear in the number of input data and thus are not directly applicable for malware analysis.

To address this problem, we propose an approximation for clustering and classification techniques inspired by the work of Bayer et al. (2009a). A set of malware binaries often contains similar variants of the same family which exhibit almost identical behavioural patterns. As a consequence, the embedded reports form dense clouds in the vector space. We exploit this dense representation by subsuming groups of similar behavior using prototypes—reports being typical for a group of homogeneous behavior. By restricting the computation of learning methods to prototypes and later propagating results to all embedded data, we are able to accelerate clustering as well as classification techniques.

The extracted prototypes correspond to regular reports and thus can be easily inspected by a human analyst, whereas the approximation of locality sensitive hashing employed by Bayer et al. (2009a) is opaque, providing almost no insights into groups of behavior.

Prototype Extraction, Clustering using Prototypes and Classification using prototypes are shown in the figures below



**Figure 2.4 Prototype Extraction, Clustering and Classification**

## 2.4 SUPPORT VECTOR MACHINES

A support vector machine (SVM) is a computer algorithm that learns by example to assign labels to objects<sup>1</sup>. For instance, an SVM can learn to recognize fraudulent credit card activity by examining hundreds or thousands of fraudulent and non-fraudulent credit card activity reports. Alternatively, an SVM can learn to recognize handwritten digits by examining a large collection of scanned images of handwritten zeroes, ones and so forth. SVMs have also been successfully applied to an increasingly wide variety of biological applications. A common biomedical application of support vector machines is the automatic classification of microarray gene expression profiles. Theoretically, an SVM can examine the gene expression profile derived from a tumor sample or from peripheral fluid and arrive at a diagnosis or prognosis. Biological applications of SVMs involve classifying objects as diverse as protein and DNA sequences, microarray expression profiles and mass spectra<sup>3</sup>.

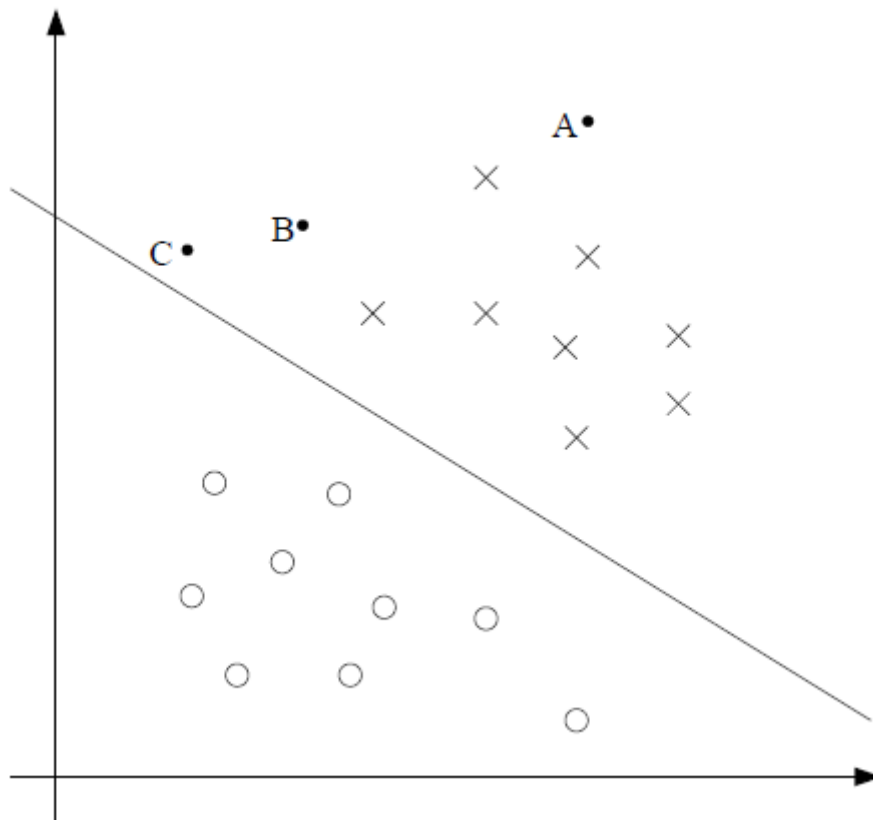
In essence, an SVM is a mathematical entity, an algorithm (or recipe) for maximizing a particular mathematical function with respect to a given collection of data. The basic ideas behind the SVM algorithm, however, can be explained without ever reading an equation. Indeed, to understand the essence of SVM classification, one needs only to grasp four basic concepts:

- (i) the separating hyperplane,
- (ii) the maximum-margin hyperplane,
- (iii) the soft margin and
- (iv) the kernel function.

Consider logistic regression, where the probability  $p(y = 1|x; \theta)$  is modelled by  $h(x) = g(\theta^T x)$ . We would then predict “1” on an input  $x$  if and only if  $h(x) \geq 0.5$ , or equivalently, if and only if  $\theta^T x \geq 0$ . Consider a positive training example ( $y = 1$ ). The larger  $\theta^T x$  is, the larger also is  $h(x) = p(y = 1|x; w, b)$ , and thus also the higher our degree of “confidence” that the label is 1. Thus, informally we can think of our prediction as being a very confident one that  $y = 1$  if  $\theta^T x \gg 0$ . Similarly, we think of logistic regression as making a very confident prediction of  $y = 0$ , if  $\theta^T x \ll 0$ . Given a training set, again informally it seems that we’d have found a good fit to the training data if we can find  $\theta$  so that  $\theta^T x(i) \gg 0$  whenever  $y(i) = 1$ , and  $\theta^T x(i) \ll 0$  whenever

$y(i) = 0$ , since this would reflect a very confident (and correct) set of classifications for all the training examples. This seems to be a nice goal to aim for, and we'll soon formalize this idea using the notion of functional margins.

For a different type of intuition, consider the following figure, in which  $x$ 's represent positive training examples,  $o$ 's denote negative training examples, a decision boundary (this is the line given by the equation  $\theta^T x = 0$ , and is also called the separating hyperplane) is also shown, and three points have also been labeled A, B and C.



Notice that the point A is very far from the decision boundary. If we are asked to make a prediction for the value of  $y$  at A, it seems we should be quite confident that  $y = 1$  there. Conversely, the point C is very close to the decision boundary, and while it's on the side of the decision boundary on which we would predict  $y = 1$ , it seems likely that just a small change to the decision boundary could easily have caused our prediction to be  $y = 0$ . Hence, we're much more confident about our prediction at A

than at C. The point B lies in-between these two cases, and more broadly, we see that if a point is far from the separating hyperplane, then we may be significantly more confident in our predictions. Again, informally we think it'd be nice if, given a training set, we manage to find a decision boundary that allows us to make all correct and confident (meaning far from the decision boundary) predictions on the training examples. We'll formalize this later using the notion of geometric margins.

## 2.5 EVALUATION METRICS

In pattern recognition and information retrieval, precision (also called positive predictive value) is the fraction of retrieved instances that are relevant, while recall (also known as sensitivity) is the fraction of relevant instances that are retrieved. Both precision and recall are therefore based on an understanding and measure of relevance.

In statistics, if the null hypothesis is that all and only the relevant items are retrieved, absence of type I and type II errors corresponds respectively to maximum precision (no false positives) and maximum recall (no false negatives).

In simple terms, high **recall** means that an algorithm returned most of the relevant results, while high **precision** means that an algorithm returned substantially more relevant results than irrelevant.

As an example, in an information retrieval scenario, the instances are documents and the task is to return a set of relevant documents given a search term; or equivalently, to assign each document to one of two categories, "relevant" and "not relevant". In this case, the "relevant" documents are simply those that belong to the "relevant" category. Recall is defined as the number of relevant documents retrieved by a search divided by the total number of existing relevant documents, while precision is defined as the number of relevant documents retrieved by a search divided by the total number of documents retrieved by that search.

In a classification task, the precision for a class is the number of true positives (i.e. the number of items correctly labeled as belonging to the positive class) divided by the total number of elements labeled as belonging to the positive class (i.e. the sum of

true positives and false positives, which are items incorrectly labeled as belonging to the class). Recall in this context is defined as the number of true positives divided by the total number of elements that actually belong to the positive class (i.e. the sum of true positives and false negatives, which are items which were not labeled as belonging to the positive class but should have been).

In information retrieval, a perfect precision score of 1.0 means that every result retrieved by a search was relevant (but says nothing about whether all relevant documents were retrieved) whereas a perfect recall score of 1.0 means that all relevant documents were retrieved by the search (but says nothing about how many irrelevant documents were also retrieved).

In a classification task, a precision score of 1.0 for a class C means that every item labeled as belonging to class C does indeed belong to class C (but says nothing about the number of items from class C that were not labeled correctly) whereas a recall of 1.0 means that every item from class C was labeled as belonging to class C (but says nothing about how many other items were incorrectly also labeled as belonging to class C).

Often, there is an inverse relationship between precision and recall, where it is possible to increase one at the cost of reducing the other. Brain surgery provides an obvious example of the tradeoff. Consider a brain surgeon tasked with removing a cancerous tumor from a patient's brain. The surgeon needs to remove all of the tumor cells since any remaining cancer cells will regenerate the tumor. Conversely, the surgeon must not remove healthy brain cells since that would leave the patient with impaired brain function. The surgeon may be more liberal in the area of the brain she removes to ensure she has extracted all the cancer cells. This decision increases recall but reduces precision. On the other hand, the surgeon may be more conservative in the brain she removes to ensure she extracts only cancer cells. This decision increases precision but reduces recall. That is to say, greater recall increases the chances of removing healthy cells (negative outcome) and increases the chances of removing all cancer cells (positive outcome). Greater precision decreases the chances of removing

healthy cells (positive outcome) but also decreases the chances of removing all cancer cells (negative outcome).

Usually, precision and recall scores are not discussed in isolation. Instead, either values for one measure are compared for a fixed level at the other measure (e.g. precision at a recall level of 0.75) or both are combined into a single measure, such as their harmonic mean the F-measure, which is the weighted harmonic mean of precision and recall (see below), or the Matthews correlation coefficient, which is the geometric mean of the regression coefficients Informedness ( $\Delta P'$ ) and Markedness ( $\Delta P$ ). Accuracy is a weighted arithmetic mean of Precision and Inverse Precision (weighted by Bias) as well as a weighted arithmetic mean of Recall and Inverse Recall (weighted by Prevalence). Inverse Precision and Recall are simply the Precision and Recall of the inverse problem where positive and negative labels are exchanged (for both real classes and prediction labels). Recall and Inverse Recall, or equivalently true positive rate and false positive rate, are frequently plotted against each other as ROC curves and provide a principled mechanism to explore operating point tradeoffs. Outside of Information Retrieval, the application of Recall, Precision and F-measure are argued to be flawed as they ignore the true negative cell of the contingency table, and they are easily manipulated by biasing the predictions. The first problem is 'solved' by using Accuracy and the second problem is 'solved' by discounting the chance component and renormalizing to Cohen's kappa, but this no longer affords the opportunity to explore tradeoffs graphically. However, Informedness and Markedness are Kappa-like renormalizations of Recall and Precision and their geometric mean Matthews correlation coefficient thus acts like a debiased F-measure.

For classification tasks, the terms true positives, true negatives, false positives, and false negatives compare the results of the classifier under test with trusted external judgments. The terms positive and negative refer to the classifier's prediction (sometimes known as the expectation), and the terms true and false refer to whether that prediction corresponds to the external judgment (sometimes known as the observation). This is illustrated by the table below:



	actual class (observation)	
	tp	fp
	(true positive) Correct result	(false positive) Unexpected result
	fn	tn
predicted class (expectation)	(false negative) Missing result	(true negative) Correct absence of result

Precision and recall are then defined as

$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

A measure that combines precision and recall is the harmonic mean of precision and recall, the traditional F-measure or balanced F-score:

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

## **CHAPTER 3**

### **WORK DONE**

#### **3.1 DATA GATHERING PHASE**

Software is considered to be malware based on the perceived intent of the creator rather than any particular features. Based on this, we have classified malware, with regard to their means of attacking a computer, as follows:-

1. Web based
  - a. Crime-ware (performing identity theft to access online accounts)
  - b. Adware (advertising-supported software)
  - c. Plug-ins
  - d. Data mines (gather data about an end user)
  - e. Exploit (take advantage of a computer's vulnerability)
  - f. Spam-ware ( capability to import thousands of email addresses )
  - g. Spyware (collects information about the user, their computer or browsing habits)
  - h. Botnet (collection of software robots)
2. External media
  - a. Computer viruses (replicates itself and infects a computer)
  - b. Baiting (uses tangible media and relies on the curiosity or greed of the victim)
  - c. Exploit
  - d. Spyware
  - e. Trojan (appears to have a normal function but actually conceals malicious functions)
3. Embedded in source code
  - a. Backdoor (circumvents regular authentication)
  - b. Rootkits (replaces essential system executables)
  - c. Spyware

### 3.2 DESIGN OF SIGNATURE MATCHING TECHNIQUES

Once the data has been gathered, this technique is used for matching existing virus signatures. Signatures are nothing but a hash of a particular piece of code of the malware. This can be done in the following ways

1. Matching of existing virus signature databases/dictionaries
  2. Analysis of data to extract previously unknown patterns
  3. Classification of data as malware or benign software
- Advantage : Signature Matching is fast since it uses pattern matching and hashing data structures
  - Disadvantage: It cannot detect viruses whose signatures are not present in the database i.e. new viruses or mutations of existing viruses.
  - Pattern matching algorithm to be used : Boyer Moore Algorithm

Key features of BM algorithm:

- A shift table is used to reduce the number of calls to the BM routine.
- Problem of tokenization
  - a. Byte-by-byte walk through the signature database
  - b. Optimization – hash of every 3 byte chunks
  - c. Most signature patterns are not a match

Customized mapping of files into memory: Since executable files being scanned for viruses are large, there is need for efficient mapping of these files into memory. We make use of a customized mapping data structure known as fmap, over the typical data structure mmap, for the following reasons:

- Faster ( Advantage for antivirus system )
- Does not use global lock for every application's map ( avoids serialization of access to map )

### 3.3 IMPLEMENTATION OF SIGNATURE BASED DETECTION

During the initial stages, we adopted the Windows environment and identified Clam AV, an open source antivirus engine which also provided an API for basic functions like loading virus databases and scanning files. However since the source code of Clam AV [Clam AV] had a native build on Linux environment, Linux environment turned out to be preferable. This led to a switch to Linux environment.

In the Linux environment, the first step included a build of the API, libclamav. Challenges faced included poor documentation of build process and figuring out dependencies.

Signature based detection aims at scanning files for virus signatures and matching them against a virus gene library.

The flow chart for the API calls is shown below

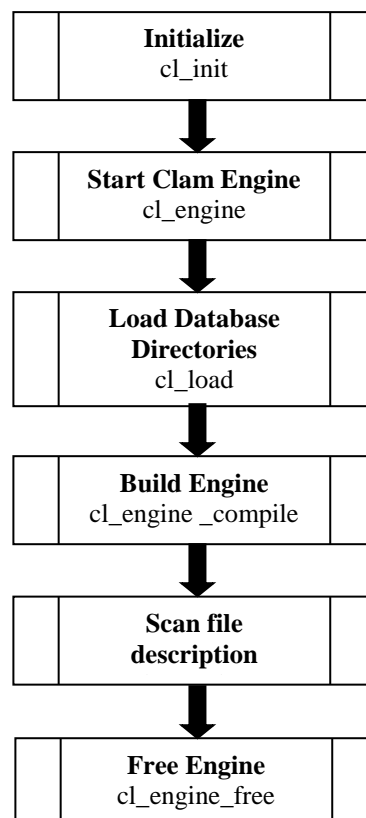
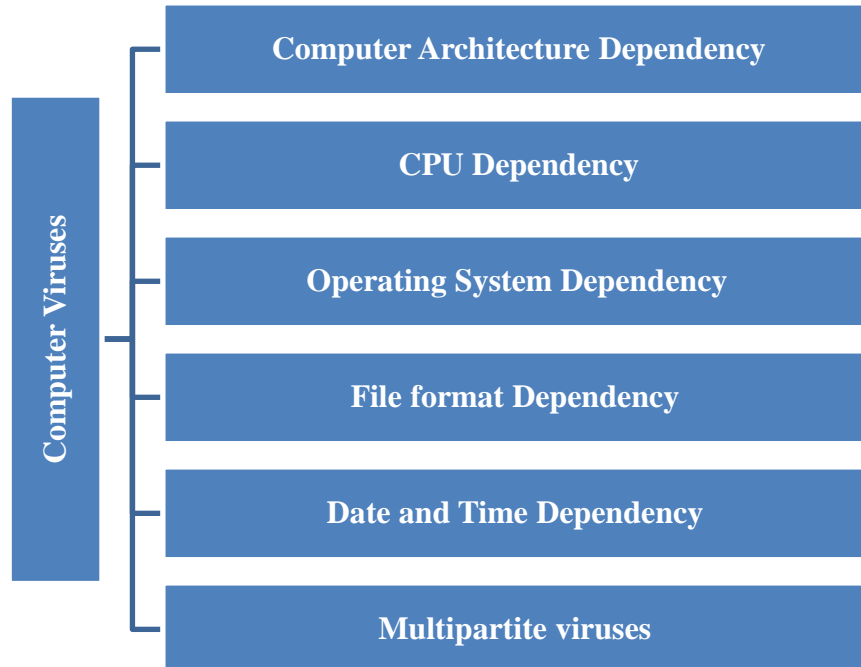


Figure 3.1: Flowchart for API function calls

### 3.4 DEPENDENCY BASED CLASSIFICATION OF VIRUSES



**Figure 3.2: Dependency based classification of viruses**

Based on the dependencies of each virus, we have come up with a classification of computer viruses that will act as a precursor to behaviour based detection of viruses.

a. Computer Architecture dependency

Most computer viruses do spread in executable, binary form (also called compiled form). For instance, a boot virus will replicate itself as a single or couple of sectors of code and takes advantage of the computer's boot sequence. In theory, it would be feasible to create a multi-architecture binary virus, but this is no simple task. It is especially hard to find ways to execute the code made for one architecture to run on another. However, it is relatively easy to code to two independent architectures, inserting the code for both in the same virus. Then the virus must make sure that the proper code gets control on the proper architecture. These viruses are known as cross platform binary viruses.

b. CPU Dependency

CPU dependency affects binary computer viruses. The source code of programs is compiled to object code, which is linked in a binary format such as an EXE (executable) file format. The actual executable contains the "genome" of a program as a sequence of instructions. The instructions consist of opcodes. Thus the sequences of bytes most likely translate to garbage code from one CPU to another because of the differences between the opcode table and the operation of the actual CPU. However, there are some opcodes that might be used as meaningful code on both systems, and some viruses might take advantage of this. Most computer viruses that are compiled to binary format will be CPU-dependent and unable to replicate on a different CPU architecture

c. Operating System Dependency

Most computer viruses can operate only on a single operating system. However, cross-compatibility between DOS, Windows, Windows 95/98, and Windows NT/2000/XP still exists on the Intel platforms even today. Thus, some of the viruses that were written for DOS can still replicate on newer systems. We tend to use latest authentic software, however, thus reducing the risk of such infections. A 32-bit Windows virus that will infect only portable executable (PE) files will not be able to replicate itself on DOS because PE is not a native file format of DOS and thus will not execute on it. However, so-called multipartite viruses are able to infect several different file formats or system areas, enabling them to jump from one operating environment to another. The most important environmental dependency of binary computer viruses is the operating system itself. Some computer viruses depend not only on a particular operating system, but also on an actual system version

d. File System Dependency

Computer viruses also have file system dependencies. For most viruses, it does not matter whether the targeted files reside on a File Allocation Table (FAT), originally used by DOS; the New Technology File System (NTFS), used by Windows NT; or a remote file system shared across network connections. For such viruses, as long as they are compatible with the operating environment's high-level file system interface, they work. They will simply infect the file or store new files on the disk without paying attention to the actual storage format. However, other kinds of viruses depend strongly on the actual file system.

e. File format Dependency

Viruses can be classified according to the file objects they can infect. Viruses such as VirDEM and Cascade only infect DOS binary files that have the COM extension. COM files do not have a specific structure; therefore, they are easy targets of viruses. A virus either appends or prepends itself to a COM file. Other viruses can infect DOS EXE files. EXE files start with a small header structure that holds the entry point of the program among other fields. EXE infector viruses often modify the entry point field of the host and append themselves to the end of the file. There are more techniques for infecting EXE files than for infecting COM files because of the format itself.

f. Date and Time Dependency

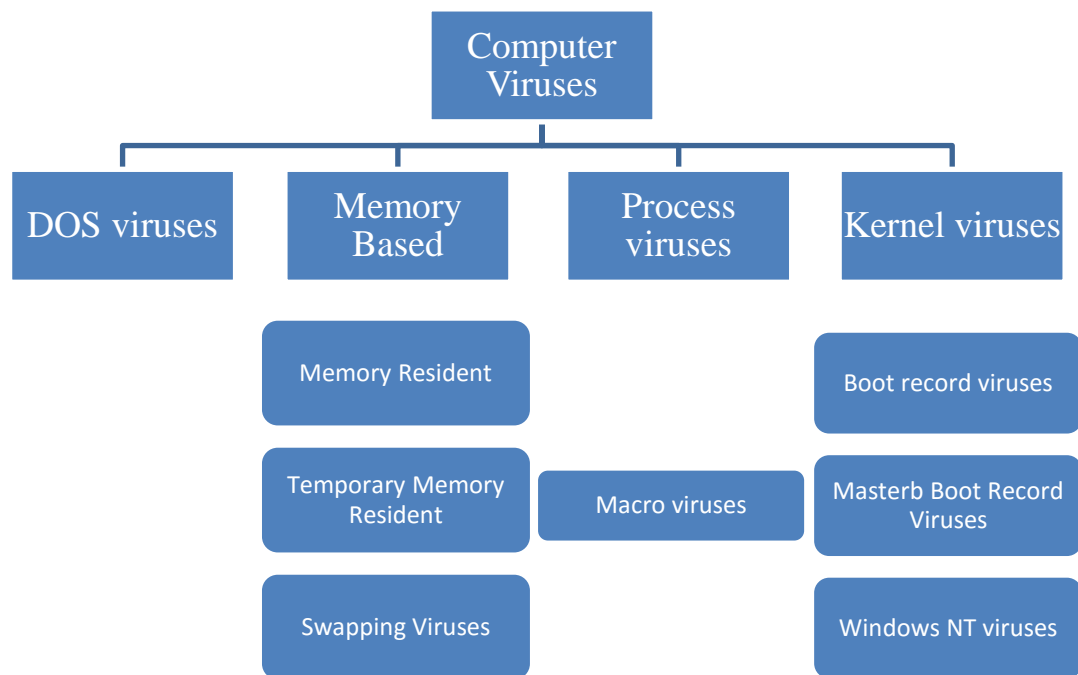
Several viruses replicate only within a certain time frame of the day. Others refuse to replicate before or after a certain date

g. Multipartite viruses

Multipartite viruses are often tricky and hard to remove. They affect boot sector and COM files. They can infect COM files on the hidden partitions that some computer manufacturers use to hide data and extra

code by marking the partition entries specifically. If the virus loads to memory before these hidden files are accessed, these files can get infected easily. Scanners typically scan the content of the visible partitions only, so such infections often lead to mysterious reinfections of the system. This is because the virus has been cleaned from everywhere but from the hidden partition, so the virus can infect the system again as soon as the hidden partition is used to run one of the infected COM files

### 3.5 BEHAVIOUR BASED CLASSIFICATION OF VIRUSES



**Figure 3.3: Classification of viruses based on behaviour**

#### 1. DOS viruses

DOS viruses are usually Direct Action viruses. Following are the characteristics of Direct Action viruses.

- Simple computer viruses
- Do not actively manifest themselves in computer memory.
- Do not spread fast and do not easily become available in the wild.



Technique of attack:

- a. Direct-action viruses load with the host program into computer memory.
- b. Upon getting control, they look for new objects to infect by searching for new files. This is exactly why one of the most common kinds of computer virus is the direct-action infector. This kind of viruses can be crafted with relative ease by the attacker on a variety of platforms, in binary or in script languages.

Mechanism of attack:

- a. Typically such viruses only infect a couple of files upon execution and use a FindFirst, FindNext sequence to look for a set of victim applications to attack.
- b. But some viruses infect everything at once by enumerating all directories for victims. In this case, direct-action viruses simply copy themselves between the diskettes and the hard disk without waiting for the user to copy an infected file to the diskette. This technique, however, makes them much more likely to be noticed by a user because the extra diskette activity is a noisy operation.

Simplest mechanism attempted by any DOS virus is either to force a program that it infects to terminate to the DOS prompt or to open another malicious file.

## 2. Memory based Viruses

Memory based viruses can be further classified as

### a. Memory Resident viruses

These are a more efficient class of computer viruses that remain in memory after the initialization of virus code. Such viruses typically follow these steps:

1. The virus gets control of the system.
2. It allocates a block of memory for its own code.

3. It relocates its code to the allocated block of memory.
4. It activates itself in the allocated memory block.
5. It hooks the execution of the code flow to itself.
6. It infects new files and/or system areas.

The vulnerability is that DOS applications are not separated or walled up from each other in any way. Malicious code can take advantage of this kind of system very easily. On standard DOS, the processor is used in a single mode, and therefore any program has the privilege to modify any other program's code in the physical memory

b. Temporary Memory Resident viruses

A slightly more intelligent type of computer virus is not always resident in the computer's memory. Instead, the virus remains in memory for a short period of time or until a particular event occurs. Such an event might be triggered after a certain number of successful infections.

Such viruses tend to be much less successful at becoming in the wild. In contrast to this, direct-action viruses are much easier to spot because they increase the disk activity considerably. Permanent resident viruses, however, are usually more infectious and spread much more rapidly than temporary memory-resident viruses.

However such viruses infect .COM files and easily spread because the disk activity is increased only when the .COM file is executed and this behaviour is expected.

c. Swapping viruses

Another technique in computer virus writing relies on loading a small piece of virus code actively into memory all the time. This small piece of code might be a hook event. Whenever the hook event is triggered, the

virus loads a segment of viral code from the disk and infects a new object. After that, the virus again clears the loaded segment from memory.

Although it appears that there are certain advantages to this technique, such as the fact that the virus consumes less physical memory and can keep its code encrypted in files most of the time, there are also many disadvantages—for instance, the possibility of introducing heavily increased disk activity that makes it much easier to spot the attack.

### 3. Process viruses

On modern, multitasking operating systems, viruses need to use slightly different strategies. The virus does not have to become resident in the memory. It is usually enough if the virus runs itself as a part of the process.

Memory space is divided according to security rings associated with the mode of the processors. Most modern operating systems, such as Windows NT-based systems, separate regular applications, which use user mode, from those that use kernel mode, such as the OS, drivers, and relevant security data structures—for better security and system stability. For this reason, applications normally do not interfere with the system kernel, as DOS programs do.

The most widely used virus for infecting via processes is the macro virus.

It is a virus that infects word processing documents or spreadsheets. Modern word processing and spreadsheet programs such as Word for Windows or Excel allow us to write simple programs, called macros, and then attach these macros to your document or spreadsheet. These macros can then be used to automate repetitive tasks, make calculations in the spreadsheet,

A macro virus is merely a malicious macro program that is designed to copy itself from document to document or spreadsheet to spreadsheet rather than serving a useful purpose. Word for Windows documents are the most common carriers for macro viruses, with well over 1,500 distinct macro virus strains.

Macro viruses are especially problematic for several key reasons. First, with increased use of the Internet, email and other workgroup software, users are exchanging more information than ever before. In the past, information exchange was not dangerous because documents and spreadsheets could not contain macros, and therefore could not contain macro viruses. Today, however, the documents most commonly exchanged through email can and do contain macro viruses.

Another factor that accounts for macro virus success is the availability of the popular office applications. Applications such as Word for Windows or Excel were designed to share the same documents and spreadsheet files regardless of where the documents originated. Therefore, a user could send a macro virus-infected document created on a PC to a co-worker that has a Macintosh. If the Macintosh user edits or views the infected document, the virus takes up residence on that system. On the other hand, traditional PC viruses will only work on the type of machine for which they were originally designed.

Macro viruses are extremely easy to construct. In the past, only programmers with low-level assembly language programming skills could create computer viruses. With the introduction of user-friendly application macros, macro viruses are more easily created.

#### 4. Kernel viruses

- a. Boot record viruses: These are typically acquired in two different ways. The first method involves booting off of an infected floppy diskette. The second method involves running a dropper program from a DOS session, which directly drops the virus onto the boot record of the active partition. Multipartite computer viruses sometimes attempt this type of infection.
- b. Master boot record viruses: These viruses use the same infection mechanism as boot record viruses except they infect the master boot record instead of the boot record. Since these days Windows NT

systems load the kernel in protection mode, affecting the master boot record in simple ways does not affect the system.

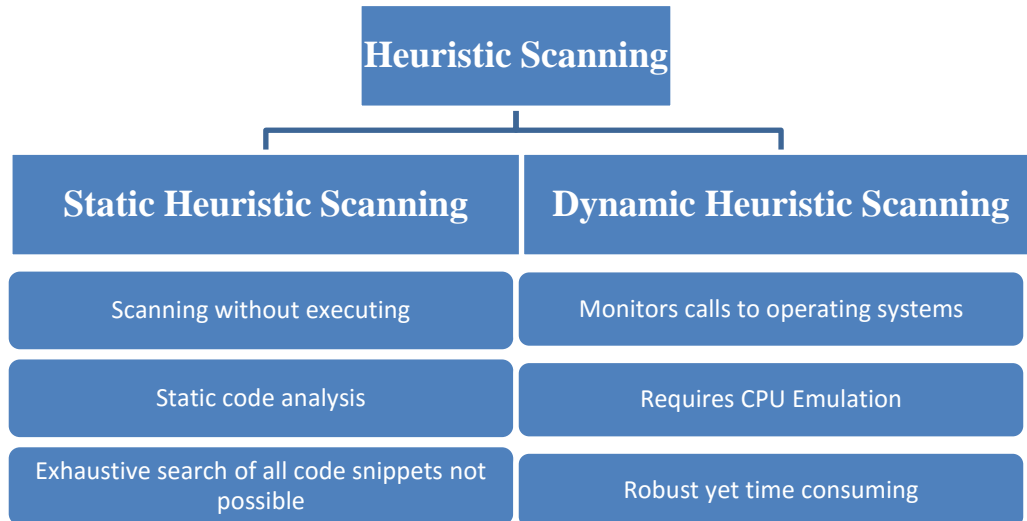
- c. Windows NT Viruses: These are memory-resident parasitic kernel-mode driver virus under Windows NT environments. The virus stays in memory as a kernel-mode driver and hooks the main NT service interrupt (INT 2Eh), so it can replicate on the transit when files are opened.

### **3.5.1 Parameters identified for heuristic scanning**

Following are the behavioral patterns we have identified to detect above listed malware

1. System calls
2. Unusual destination
3. Analysis of file types and file system
4. Analysis of memory usage – buffer overflow analysis, system registry
5. Access of executables, mutex
6. Access of disk
7. Replication
8. Attempts to hide other files
9. Source code content matched using wild card characters
10. Attempts to terminate programs
11. Attempts to open other executables.

### 3.6 IMPLEMENTATION OF HEURISTIC SCANNING



**Figure 3.4: Classification of Heuristic scanning based on approach**

#### 3.6.1 Implementation details of static heuristic scanning of DOS viruses

As explained earlier, DOS viruses are those viruses which infect the system by hooking onto the DOS kernel. Based on our study, we realized that DOS virus behaviour is indicated by the following behavioural patterns

- If a program is trying to terminate illegitimately to the DOS prompt, there are chances that this program is a DOS virus
- If a program is trying to open another executable and attempting to execute it on the DOS prompt, there are chances that this program is a DOS virus.

In static heuristic scanning, these behavioural patterns are detected based on the presence of machine language instructions to perform the above mentioned tasks. However each one of those tasks can be performed in multiple ways and hence will have different machine language code patterns. For eg, consider a program that is trying to exit to the DOS prompt. Given below are the machine language bytes and the assembly level code.

Machine Language Bytes (Hexadecimal)	Assembly level code
B8 00 4C	MOV AX, 4C00
CD 21	INT 21H

**Figure 3.5: Machine Language and Assembly code for terminating a program – permutation 1**

However this is not the only way in which a program can try to exit to the DOS prompt. Another mechanism is show below.

Machine Language Bytes (Hexadecimal)	Assembly Level Code
B4 3C	MOV AH,3C
BB 00 00	MOV BX,0000
88 D8	MOV AL,BL
80 C4 10	ADD AH,10
8E C3	MOV ES, BX
9C	PUSHF
26	ES:
FF 1E 84 00	CALL FAR [0084]

**Figure 3.6: Machine Language and Assembly code for terminating a program – permutation 2**

Thus there is a need to identify as many byte sequences as possible. Following are few of the byte sequences that we have identified for termination of a program

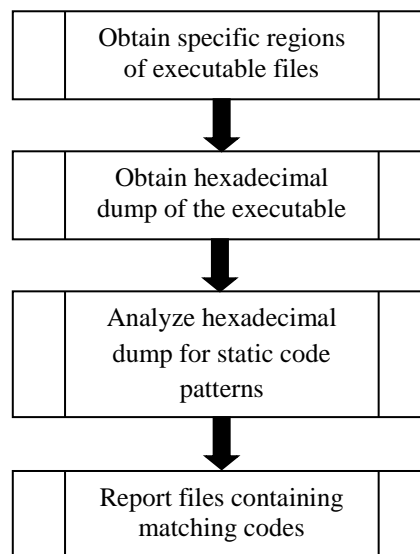
1. B8 ?? 4C CD 21
2. B4 4C CD 21
3. B4 4C B0 ?? CD 21
4. B0 ?? B4 4C CD 21

Similarly following are the byte sequences we have identified for a program that attempts to open a file

1. B8 02 3D BA ?? ?? CD 21
2. BA ?? ?? B8 02 3D CD 21

Also since DOS viruses have a specific characteristic of appending themselves either to the end of a .exe file or beginning or end of .com files, the executable needs to be scanned only in specific regions, thus reducing the amount of hexadecimal code to be scanned.

The steps involved in detecting the behavioural patterns of an executable file are



**Figure 3.7: Flowchart for static heuristic scanning**

### **3.6.2 Obfuscation Techniques**

Work completed includes identification of various obfuscation techniques possible and ways to handle them[1,2]. Code obfuscation techniques that we plan to target are –

a. **Dead code insertion**

Also known as trash insertion, dead-code insertion adds code to a program without modifying its behaviour. Inserting a sequence of nop instructions is the simplest example. More interesting obfuscations involve constructing challenging code sequences that modify the program state, but restore it immediately. Code sequences that are complicated enough make automatic analysis very time consuming. For example, passing values through memory rather than registers or the stack requires accurate pointer analysis to recover



values. Not all dead-code sequence can be detected and eliminated, as this problem reduces to program equivalence (i.e., is the code sequence equivalent to an empty program?), which is not decidable.

Original code	Obfuscated code
call 0h	call 0h
pop ebx	pop ebx
push ecx	nop
push eax	push ecx
pop ebx	inc ecx
add ebx, 1Ch	push eax
mov ebp, ebx	dec [esp - 0h]
	dec ecx
	pop ebx
	add ebx, 1Ch
	mov ebp, ebx

b. Code transposition

Original code	Obfuscated using code transposition
call 0h	call 0h
pop ebx	pop ebx
push ecx	jmp s2
push eax	s3:push eax
pop ebx	push eax
add ebx, 1Ch	jmp s4
mov ebp, ebx	add ebx, 1Ch
	jmp s6
	s2:push ecx
	jmp s3
	s4:pop ebx
	jmp s5
	s5:mov ebp, ebx

Code transposition shuffles the instructions so that the order in the binary image is different from the execution order, or from the order of instructions

assumed in the signature used by the antivirus software. The first variation randomly reorders the instructions and inserts unconditional branches or jumps to restore the original control-flow. The second variation swaps instructions if they are not interdependent, similar to compiler code generation, but here the goal of randomizing is not to optimize. This technique is more complicated as it requires us to ascertain code interdependency.

c. Register reassignment

The register reassignment transformation replaces usage of one register with another in a specific range. This technique exchanges register names and has no other effect on program behavior. For example, if register ebx is unused throughout a given range of the register eax's usage, it can replace eax in that range. De-obfuscation is challenging as it requires us to ascertain all possible register replacements.

d. Instruction substitution

Original code	Obfuscated code
call 0h	call 0h
pop ebx	pop ebx
push ecx	sub esp, 03h
push eax	add [esp], 1Ch
pop ebx	mov ebx, [esp]
add ebx, 1Ch	inc esp
mov ebp, ebx	mov ebp, ebx

In instruction substitution, a set of instructions are substituted by another set of instructions performing the same functionality. Static code analysis techniques without obfuscation handling will not be able to detect these obfuscated codes.

### 3.6.3 Obfuscation Handling

#### a. Dead code insertion

Dead code insertion such as insertion of nop statements can be handled in static code analysis by inserting the hexadecimal machine code for the nop statement at the end of each and every instruction any number of times.

For eg. If the signature of a malicious file is

```
E800 0000 005B 8D4B 4251 5050
0F01 4C24 FE5B 83C3 1CFA 8B2B
```

The signature of the code obfuscated with nop-insertion is

```
E800 0000 005B 8D4B 4290 5150
5090 0F01 4C24 FE5B 83C3 1C90
FA8B 2B
```

Thus to identify the malware efficiently i.e. detect insertion of nop anywhere in the code, the pattern to be searched for should be

```
E800 0000 00(90)* 5B(90)*
8D4B 42(90)* 51(90)* 50(90)*
50(90)* 0F01 4C24 FE(90)*
5B(90)* 83C3 1C(90)* FA(90)*
8B2B
```

#### b. Code Transposition

Code Transposition can be handled if we have a technique that determines if the end result is the same irrespective of the flow of the control. Hence to handle obfuscation via code transposition, we make use of an annotator known as the Control Flow Graph (CFG). The CFG of a non-obfuscated code is generated which becomes the base automaton. The obfuscated code is now passed as input to the CFG. If the execution of all the statements in the

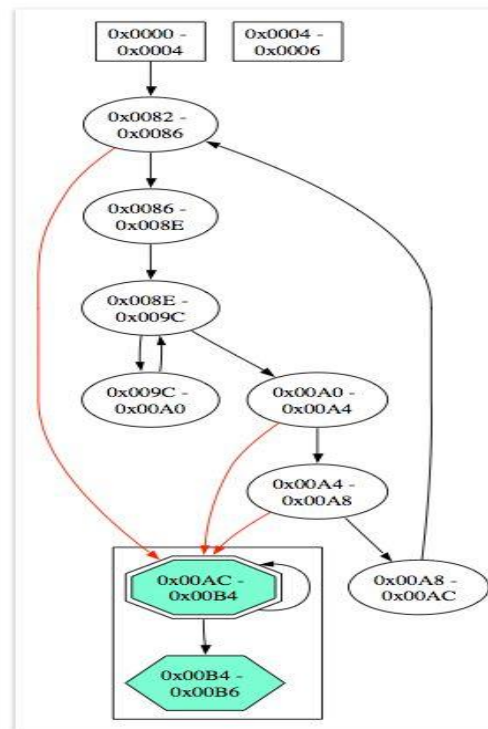
obfuscated code results in the same final state of the CFG as that of the non-obfuscated code then the code can be declared to be malicious. A sample CFG is shown below in Figure 3.8 and its textual form is shown in Figure 3.9

### c. Register swap

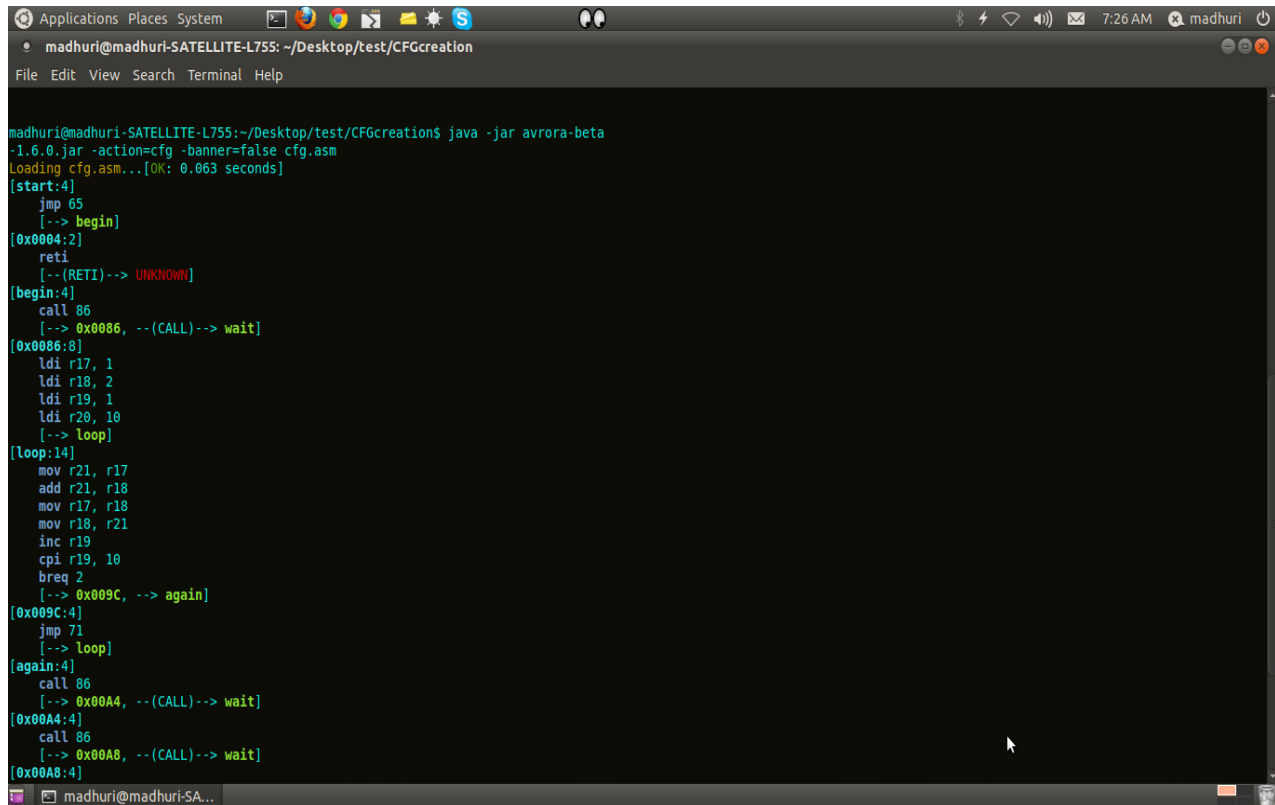
Register swap can be handled by replacing register values in hexadecimal dump values with wild card characters in matching the expression to detect malicious activity. This can be done as shown below

**B8 ?? 4C CD 21**

Where ?? indicates the register values. Hence swapping of registers will still continue to get detected as malicious code.



**Figure 3.8: Control Flow Graph**



```
madhuri@madhuri-SATELLITE-L755: ~/Desktop/test/CFGcreation$ java -jar avrora-beta
-1.6.0.jar -action=cfg -banner=false cfg.asm
Loading cfg.asm...[OK: 0.063 seconds]
{start:4}
  jmp 65
  [-> begin]
{0x0004:2}
  reti
  [--(RETI)--> UNKNOWN]
{begin:4}
  call 86
  [-> 0x0086, --(CALL)--> wait]
{0x0086:8}
  ldi r17, 1
  ldi r18, 2
  ldi r19, 1
  ldi r20, 10
  [-> loop]
{loop:14}
  mov r21, r17
  add r21, r18
  mov r17, r18
  mov r18, r21
  inc r19
  cpi r19, 10
  breq 2
  [-> 0x009C, --> again]
{0x009C:4}
  jmp 71
  [-> loop]
{again:4}
  call 86
  [-> 0x00A4, --(CALL)--> wait]
{0x00A4:4}
  call 86
  [-> 0x00A8, --(CALL)--> wait]
{0x00A8:4}
```

Figure 3.9 Control Flow Graph in textual format

#### d. Code Substitution

Algorithmic detection of code substitution will not cover all possible substitutes for a given line of code. Hence code substitution requires the obfuscated code to be run so that essential parameters can be monitored and malicious activity can be detected.

### 3.6.4 Dynamic Heuristic Scanning

Since static heuristic scanning cannot handle all obfuscation techniques like code substitution, dynamic heuristic scanning is deployed on the malware binaries. In dynamic heuristic scanning, malware binaries are allowed to run in a sandboxed environment and their behaviour is monitored based on the parameters mentioned in section 3.5.1

The following software applications can be used to implement dynamic heuristic scanning

1. Anubis[Anubis 2003]

Anubis is a service for analyzing malware. Submit your Windows executable and receive an analysis report telling you what it does. Alternatively, submit a suspicious URL and receive a report that shows you all the activities of the Internet Explorer process when visiting this URL.

2. Sunbelt CWSandbox [Sunbelt 2009]

CWSandbox uses API hooking to dynamically analyze the malware. While the report won't be as contextually easy to understand as the reports provided by ThreatExpert, the CWSandbox report contains more of the (sometimes extraneous) modifications made by the malware. CWSandbox supports more filetypes than ThreatExpert, with new types like XML representation of reports constantly being added.

3. Joebox[Joebox 2008]

Joebox also uses API hooking and supports script as well as dynamic link library (DLL) file and executables. A script and binary can be packaged together during the upload so that the interaction between the two is observed.

4. Sandboxie [Sandboxie 2005]

Sandboxie runs your programs in an isolated space which prevents them from making permanent changes to other programs and data in your computer. It offers secure web browsing by running your Web browser under the protection of Sandboxie means that all malicious software downloaded by the browser is trapped in the sandbox and can be discarded trivially. It has enhanced privacy where browsing history, cookies, and cached temporary files collected while Web browsing stay in the sandbox and don't leak into Windows. It prevents wear-and-tear in Windows by installing software into an isolated sandbox.

## 5. Cuckoo Sandbox[25]

Cuckoo is an open source automated malware analysis system. It automatically runs and analyzes files and generates results that outline what the files do while running in an isolated Windows Operating System. Due to its extremely modular design, Cuckoo can be used both as a standalone application as well as to be integrated in larger frameworks.

The key features indicated in the results include:

- Traces of win32 API calls performed by all processes spawned by the malware.
- Files being created, deleted and downloaded by the malware during its execution.
- Memory dumps of the malware processes.
- Network traffic trace in PCAP format.
- Screenshots of Windows desktop taken during the execution of the malware.
- Full memory dumps of the machines.

The files that can be analyzed include Generic Windows executables, DLL files, PDF documents, Microsoft Office documents, URLs, PHP scripts and any other executable in Windows operating system.

Out of the above mentioned software applications CWSandbox was used for sandboxing the malware binaries. This is because only CWSandbox provides the report in XML formats which are easy to understand, parse and convertible to other forms that may be convenient for parsing by the next module.

### **3.7 MACHINE LEARNING**

While dynamic heuristic scanning overcomes code obfuscation, it also creates the problem of false positives as it is behaviour based detection. With an aim to reduce the number of false positives, the machine learning module is introduced.

The machine learning technique being used is known as Case-based Reasoning (CBR). In summary, CBR is the process of solving new problems based on the solutions of similar past problems. It is a prominent kind of analogy making

At first, CBR may seem similar to rule induction algorithms in machine learning. Similar to a rule induction algorithm, CBR starts with a set of cases or training examples; forms a generalization of these examples by identifying commonalities between a retrieved case and the target problem.

The key difference however between the implicit generalization in CBR and the generalization in rule induction algorithms lies in when the generalization is made. A rule-induction algorithm draws its generalizations from a set of training examples before the target problem is even known; that is, it performs eager generalization. The difficulty for the rule-induction algorithm is in anticipating the different directions in which it should attempt to generalize its training examples. This is in contrast to CBR, which delays (implicit) generalization of its cases until testing time – a strategy of lazy generalization. Thus it can generalize its cases exactly as needed to cover this situation. CBR therefore tends to be a good approach for rich, complex domains in which there are multiple ways to generalize a case.

However, without statistically relevant data for backing and implicit generalization, there is no guarantee that the generalization is correct.

Two concepts for such automatic analysis of behaviour based on machine learning techniques have been recently proposed:

(a) Clustering of behaviour - aims at discovering novel classes of malware with similar behaviour (Bailey et al., 2007; Bayer et al., 2009a) and



(b) Classification of behaviour - enables assigning unknown malware to known classes of behaviour (Lee and Mody, 2006; Rieck et al., 2008).

Previous work has studied these concepts as competing paradigms, where either one of the two has been applied using different algorithms and representations of behaviour.

We have developed a framework where clustering and classification of malware binaries are used as complementary paradigms. In particular it supports the following:

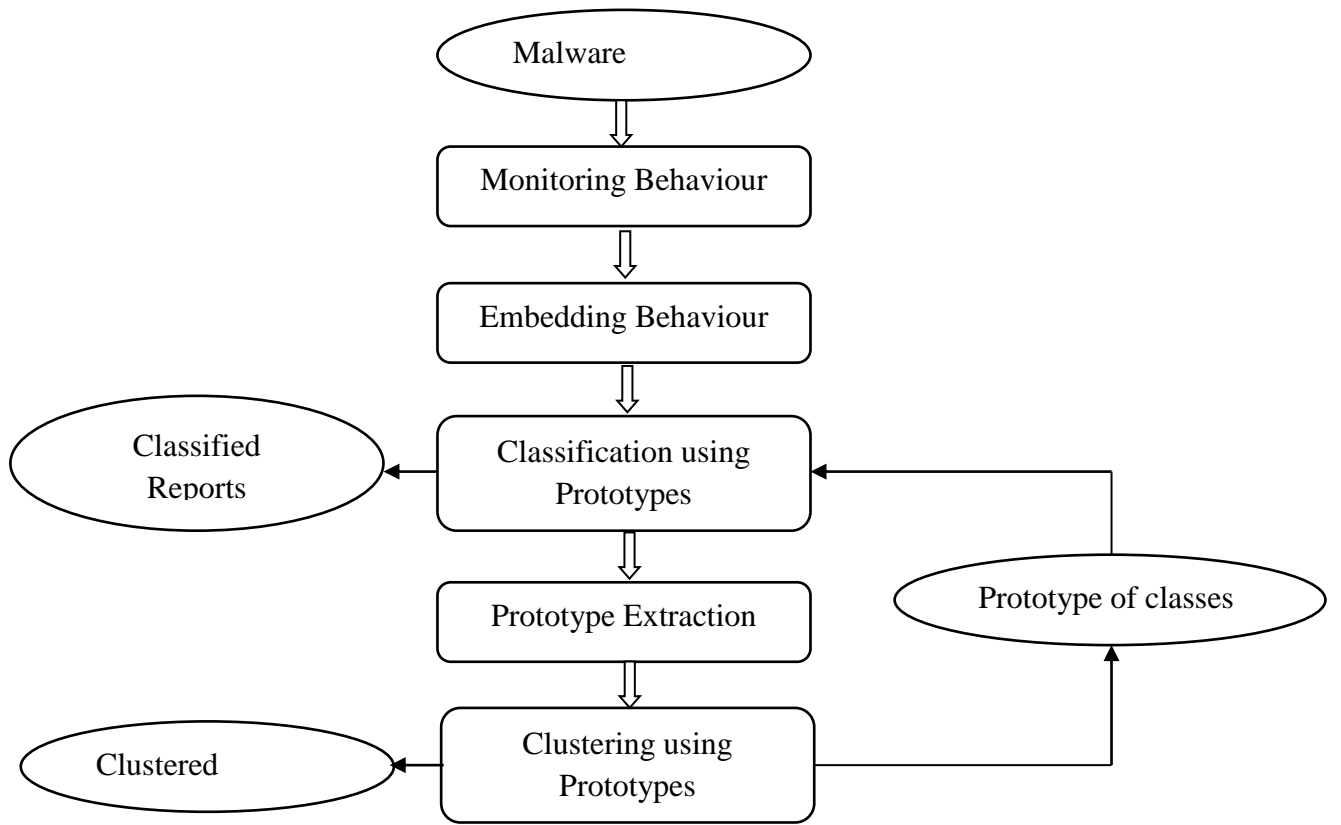
- *Scalable clustering and classification* - We propose a mapping of monitored behaviour to a vector space, such that behavioural patterns are efficiently accessible to means of machine learning. To attain scalable computation with thousands of vectors, we introduce an approximation using prototype vectors which is applicable to clustering as well as classification techniques.
- *Incremental analysis of malware behaviour* - By combining clustering and classification, we devise an incremental approach to behaviour-based analysis capable of processing the behaviour of thousands of malware binaries on a daily basis. This incremental analysis significantly reduces the run-time and memory overhead of batch analysis methods, while providing accurate discovery of novel malware.
- *Extensive evaluation with real malware* – The framework has been evaluated extensively against malware binary reports collected from CWSandbox.

### **3.7.1 Schematic overview of framework**

The basic analysis steps of the framework are summarized below and shown in Figure 3.10

- a. Reports from CWSandbox are collected in XML formats. Based on the performed operations and actions—in terms of system calls—CWSandbox generates a sequential report of the monitored behaviour for each binary, where system calls and their arguments are stored in a representation specifically tailored to behaviour-based analysis

- b. The sequential reports are then embedded in a high-dimensional vector space using a library called malheur, where each dimension is associated with a behavioural pattern, a short sequence of observed instructions. In this vector representation, the similarity of behaviour can be assessed geometrically, which allows for designing intuitive yet powerful clustering and classification methods.
- c. Machine learning techniques for clustering and classification are then applied to the embedded reports for identifying novel and known classes of malware. Efficient computation of both analysis techniques is realized using prototype vectors which take larger groups of reports with similar behaviour and thereby provide an effective approximation to exact analysis.
- d. By alternating between clustering and classification steps, the embedded behaviour of malware can be analysed incrementally. First, behaviour matching known malware classes is identified using prototype vectors of previously discovered clusters. Then reports with unidentified behaviour are clustered for discovery of novel malware classes.



**Figure 3.10 Framework work flow**

### 3.7.2 Clustering and Classification

We use the concept of prototypes to achieve clustering and classification in a scalable manner. Features are extracted from the reports in the form of feature vectors using the n-gram technique. The embedding function is  $\phi$

#### 3.7.2.1 Prototype Extraction

Extracting a small yet representative set of prototypes from a data set is not a trivial task. Most approaches for prototype extraction rest on clustering (Bezdek and Kuncheva, 2001) or super-linear computations (Harmeling et al., 2006), and thus are inappropriate as basis for efficient approximation. Furthermore, the task of finding an optimal set of prototypes can be shown to be NP-hard (Garey and Johnson, 1979). Fortunately, we can adapt a linear-time algorithm by Gonz`alez (1985) which provably determines a set of prototypes only twice as large as the optimal solution. The algorithm is shown below

```

1: prototypes  $\leftarrow \varphi$ 
2: distance[x]  $\leftarrow \infty$  for all  $x \in \text{reports}$ 
3: while max(distance) > dp do
4:   choose z such that distance[z] = max(distance)
5:   for  $x \in \text{reports}$  and  $x \neq z$  do
6:     if distance[x] > ||  $\varphi(x) - \varphi(z)$  || then
7:       distance[x]  $\leftarrow$  ||  $\varphi(x) - \varphi(z)$  ||
8:   add z to prototypes

```

The algorithm proceeds by iteratively selecting prototypes from a set of reports. The first prototype is either fixed or chosen at random. During each run, the distance from the current set of prototypes to the remaining embedded reports is computed (line 5–7). The farthest report is chosen as new prototype, such that the data set is iteratively covered by a web of prototypes (line 4). This procedure is repeated until the distance from each vector to its nearest prototype is below the parameter dp (line 3).

The run-time complexity of this algorithm is  $O(kn)$  where  $n$  is the number of reports and  $k$  the number of prototypes. Given that dp is reasonably chosen, the algorithm is linear in the number of reports, as  $k$  solely depends on the distribution of data.

### 3.7.2.2 Clustering using Prototypes

Clustering refers to a basic technique of machine learning which aims at partitioning a given data set into meaningful groups, so called clusters. The partitioning is determined, such that objects within one cluster are similar to each other, whereas objects in different clusters are dissimilar. Clustering enables discovery of structure in unknown data and thus has been employed in a large variety of applications.

Clustering for analysis of malware behaviour has been proposed by Bailey et al. (2007) and later refined by Bayer et al. (2009a). We pursue this line of research and study the standard technique of hierarchical clustering (Duda et al., 2001) for determining groups of malware with similar behaviour. In contrast to previous work, we base our analysis on the concept of prototypes. That is, first clusters of prototypes

are determined and then propagated to the original data. The clustering algorithm is as shown below

```

1: for  $z, z_0 \in \text{prototypes}$  do
2:    $\text{distance}[z, z_0] \leftarrow ||\phi(z) - \phi(z_0)||$ 
3: while  $\min(\text{distance}) < dc$  do
4:   merge clusters  $z, z_0$  with minimum  $\text{distance}[z, z_0]$ 
5:   update distance using complete linkage
6: for  $x \in \text{reports}$  do
7:    $z \leftarrow \text{nearest prototype to } x$ 
8:   assign  $x$  to cluster of  $z$ 
9: reject clusters with less than  $m$  members

```

Starting with each prototype being an individual cluster, the algorithm proceeds by iteratively determining and merging the nearest pair of clusters (line 4). This procedure is terminated if the distance between the closest clusters is larger than the parameter  $dc$ . To compute distances between clusters, the algorithm considers the maximum distance of their individual members—a standard technique of hierarchical clustering referred to as complete linkage (Anderberg, 1973; Duda et al., 2001). Once a clustering has been determined on the prototypes, it is propagated to the original reports (line 6–8). Moreover, clusters with fewer than  $m$  members are rejected and kept for later incremental analysis.

The algorithm has a run-time complexity of  $O(k^2 \log k + n)$ , where  $n$  is the number of reports and  $k$  the number of prototypes. In comparison to exact hierarchical clustering with a run-time of  $O(n^2 \log n)$ , the approximation provides a speed-up factor of square root  $n/k$ .

### 3.7.2.3 Classification using Prototypes

We next consider classification, which allows to learn a discrimination among different classes of objects. Classification methods require a learning phase prior to application, where a model for discrimination is inferred from a data set of labelled objects. This model can then be applied for predicting class labels on unseen data. As many real-world application fall into this concept of learning, a large body of research exists on designing and applying classification methods (Mitchell, 1997; Duda et al., 2001; Müller et al., 2001; Shawe-Taylor and Cristianini, 2004)

The application of classification for the analysis of malware behaviour has been studied by Lee and Mody (2006) and Rieck et al. (2008). In both approaches, the behaviour of unknown malware is classified to known classes of behaviour, where the initial training data is labelled using anti-virus scanners. Unfortunately, most anti-virus products suffer from inconsistent and incomplete labels (Bailey et al., 2007) and do not provide sufficiently accurate labels for training. As a remedy, we employ the malware classes discovered by clustering as labels for training and thereby learn a discrimination between known clusters of malware behaviour. As these clusters are represented by prototypes in our framework, we again make use of approximation to accelerate learning.

The algorithm is given below –

```

1: for  $x \in \text{reports}$  do
2:    $z \leftarrow \text{nearest prototype to } x$ 
3:   if  $\| \varphi(z) - \varphi(x) \| > dr$  then
4:     reject  $x$  as unknown class
5:   else
6:     assign  $x$  to cluster of  $z$ 

```

For each report  $x$ , the algorithm determines the nearest prototype of the clusters in the training data (line 1–2). If the nearest prototype is within the radius  $dr$ , the report is assigned to the respective cluster, whereas otherwise it is rejected and hold back for later incremental analysis (line 4–6). This procedure is referred to as nearest prototype classification and resembles an efficient alternative to costly  $k$ -nearest neighbour methods (Bezdek and Kuncheva, 2001; Duda et al., 2001).

The run-time of the algorithm is  $O(kn)$ , as for each of the  $n$  reports the nearest prototype needs to be determined from the  $k$  prototypes in the training data. It also has scope for parallelization on multi-core systems.

### 3.7.3 Incremental Analysis

Based on combining of clustering and classification techniques, we devise an incremental approach to analysis of malware behaviour. While previous algorithms have been restricted to batch analysis, incremental analysis processes the incoming reports of malware behaviour in small chunks. To implement an incremental analysis, we need to keep track of intermediate results, such as clusters determined during previous runs of the algorithm. This is where the concept of prototypes enables us to store discovered clusters in a concise representation and, moreover, provides a significant speed-up if used for classification.

The incremental analysis algorithm is given below :

```
1: rejected  $\leftarrow \varnothing$ , prototypes  $\leftarrow \varnothing$ 
2: for reports  $\leftarrow$  data source  $\cup$  rejected do
3: classify reports to known clusters using prototypes
4: extract prototypes from remaining reports
5: cluster remaining reports using prototypes
6: prototypes  $\leftarrow$  prototypes  $\cup$  prototypes of new clusters
7: rejected  $\leftarrow$  rejected reports from clustering
```

In the first processing phase, the incoming reports are classified using prototypes of known clusters (line 3). Thereby, variants of known malware are efficiently identified and filtered from further analysis. In the following phase, prototypes are extracted from the remaining reports and subsequently used for clustering of behaviour (line 4–5). The prototypes of the new clusters are stored along with the original set of prototypes, such that they can be applied in a next run for classification. This procedure—alternating between classification and clustering is repeated incrementally, where the amount of unknown malware is continuously reduced and the prevalent classes of malware are automatically discovered.

The run-time of the incremental algorithm is  $O(nm + k^2 \log k)$  for a chunk of  $n$  reports, where  $m$  is the number of prototypes stored from previous runs and  $k$  the number of prototypes extracted in the current run. Though the run-time complexity is quadratic in  $k$ , the number of extracted prototypes during each run remains constant

for chunks of equal size and distribution. Thus, the complexity of incremental analysis is determined by  $m$ , the number of prototypes for known malware classes.

### **3.8 EVALUATION**

#### **3.8.1 Evaluation Data**

A reference data set containing known classes of malware is used to evaluate and calibrate our framework.

The reference data set is extracted from a large database of malware binaries' reports maintained at the CWSandbox web site. From the overall database, we select binaries which have been assigned to a known class of malware. We discard classes with less than 20 samples and restrict the maximum contribution of each class to 300 binaries. The selected malware binaries are then executed and monitored using CWSandbox, resulting in a total of 3,133 behaviour reports in MIST format.

#### **Malware class – 24 classes**

- a. ADULTBROWSER 262
- b. ALLAPPLE\_ 300
- c. BANCOS 48
- d. CASINO 140
- e. DORFDO 65
- f. EJIK 168
- g. FLYSTUDIO 33
- h. LDPINCH 43
- i. LOOPER 209
- j. MAGICCASINO 174
- k. PODNUHA 300
- l. POSION 26
- m. PORNDIALER 98
- n. RBOT 101
- o. ROTATOR\_ 300
- p. SALITY 85
- q. SPYGAMES 139
- r. SWIZZOR 78
- s. VAPSUP 45
- t. VIKINGDLL 158



- u. VIKINGDZ 68
- v. VIRUT 202
- w. WOIKOINER 50
- x. x ZHELATIN 41

### 3.8.2 Performance measure

To assess the performance of the components, namely prototype extraction, clustering using prototype and classification using prototype, we employ the evaluation metrics of precision and recall.

The precision  $P$  reflects how well individual clusters agree with malware classes and the recall  $R$  measures to which extent classes are scattered across clusters. Formally, we define precision and recall for a set of clusters  $C$  and a set of malware classes  $Y$  as

$$P = \frac{1}{n} \sum_{c \in C} \#_c \quad \text{and} \quad R = \frac{1}{n} \sum_{y \in Y} \#_y$$

Where,

$\#_c$  is the largest number of reports in cluster  $c$  sharing the same class and

$\#_y$  the largest number of reports labelled  $y$  within one cluster.

If each report is represented as a single cluster, we obtain  $P = 1$  with low recall, whereas if all reports fall into the same cluster, we have  $R = 1$  with low precision. We consider an aggregated performance score for our evaluation, denoted as F-measure, which combines Precision and Recall

$$F = \frac{2 \cdot P \cdot R}{P + R}$$

## CHAPTER 4

### RESULTS AND ANALYSIS

#### 4.1 ANALYSIS OF FRAMEWORK

	Signature Matching	Static Heuristic Scanning	Dynamic Heuristic Scanning	Machine Learning
<b>Detection of known virus</b>	✓	✓	✓	✓
<b>Detection of unknown viruses</b>	Fails when signature is unavailable	✓	✓	62%
<b>Robustness</b>	Fails when signature is unavailable	✓	✓✓	✓
<b>False positives</b>	No false positives	✓	✓	Reduces false positives
<b>High speed detection</b>	✓✓	✓✓	Requires CPU emulation	Learning algorithms consume time
<b>Detect metamorphic/oligomorphic viruses*</b>	Fails since virus encrypts itself	✓	✓✓	Efficient only after detection by heuristic scanning
<b>Obfuscation</b>	Fails since virus obfuscates itself	✓	✓✓	Only after heuristic scanning

\* Metamorphic viruses change their body – wild card characters are used to detect patterns, Oligomorphic viruses encrypt their body – decryption routines are to be detected in code

Figure 4.1: Comparison of various artificial intelligence techniques

Based on the comparison table shown above, it is clear that a combination of all the above mentioned techniques will help achieve the best protection. Signature matching, though fails to identify unknown viruses, is retained in the sequence flow as it will be highly time consuming if heuristic scanning alone is applied to all the files on a system. Hence the detection search space is first reduced by detecting all

viruses whose signatures are available in the database using signature matching as it is fast.

Static heuristic scanning achieves almost all the requirements of an antivirus system. However, it results in false positives and is not as robust as dynamic heuristic scanning. Dynamic heuristic scanning alone cannot be used as it involves CPU emulation which is highly time consuming. Hence only those files which pass the static heuristic scanning phase and have high indications of viral behaviour are emulated on the CPU in dynamic scanning phase. Artificial Learning can be used to reduce the rate of false positives.

The implementation of static heuristic scanning has been done for DOS viruses and memory based viruses. The detection has been robust when it comes to viruses attempting to terminate a program to exit to the DOS prompt and viruses attempting to open other files. However false positives like programs attempting to make a legitimate termination to DOS prompt are being detected and need to be reduced with the aid of CPU emulation and artificial learning.

Oligomorphic viruses require dynamic heuristic scanning. Oligomorphic viruses are those that encrypt themselves. The static heuristic scanning method for oligomorphic viruses is to look for decryption schemes in the code during static code analysis. Decryption scheme detection cannot be done by pattern matching alone. Dynamic code analysis or behaviour based detection is required. This technique makes the framework more robust but also generates false positives.

In order to reduce false positives, machine learning algorithms are used. Machine Learning ensures proactive protection and reduction of false positives. It helps detect new viruses based on cases that it has already learnt.

## 4.2 F-MEASURE ANALYSIS OF LEARNING

Technique	F-measure
<b>Clustering using Prototype</b>	0.950
<b>Classification using Prototype</b>	0.981
<b>Classification using SVM and XML</b>	0.807

**Figure 4.2 F-measure of various classification techniques**

The training set used here consisted of 3,133 malware binary reports in the MIST (Malware Instruction Set) format. The machine learning component of the framework is evaluated based on 2 metrics, Precision and Recall. In a classification task,

- Precision for a class - the number of true positives (i.e. the number of items correctly labeled as belonging to the positive class) divided by the total number of elements labeled as belonging to the positive class (i.e. the sum of true positives and false positives, which are items incorrectly labeled as belonging to the class).
- Recall - the number of true positives divided by the total number of elements that actually belong to the positive class (i.e. the sum of true positives and false negatives, which are items which were not labeled as belonging to the positive class but should have been).

Precision and Recall are inversely related. If one of the metrics is to be improved, the other gets compromised. Hence to evaluate the performance of a classifier, we use another measure known as F-measure given by

$$F = \frac{2 \cdot P \cdot R}{P + R}$$

An F-measure of 1 implies a perfect classification, while an F-measure of 0 implies a completely incorrect classification.

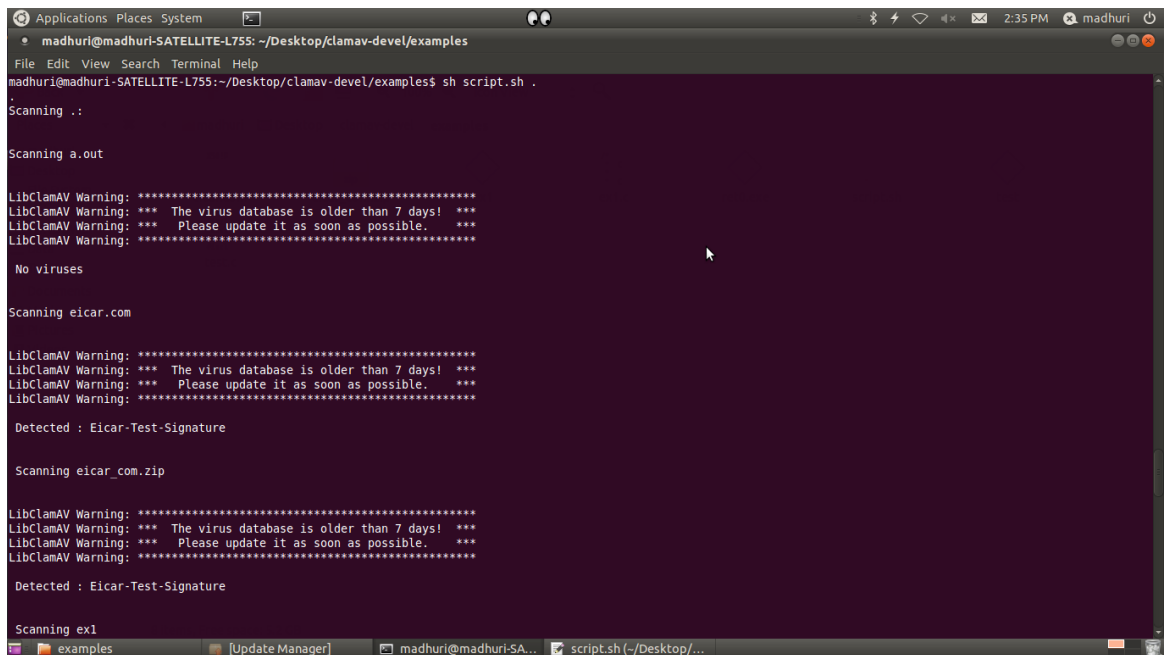
F-measure for the 3 phases in the machine learning algorithm-

- Prototype extraction occurs with a precision of 0.95.
- Classification results in a precision between 0.96 and 0.99, resulting in an F-measure of 0.981.
- A standard SVM ( Support Vector Machine ) classifier is executed on the XML reports of malware binaries sandboxed, which results in an F-measure of 0.807, thereby showing that our framework has yielded a better F-measure. It is due to the following reasons –
  - Embedding of reports from XML format into MIST ( Malware Instruction Set ) format.
  - Using clustering and classification as complimentary paradigms in incremental analysis i.e. use of Case based reasoning.

Clustering and Classification require us to decide parameters like minimum distance between clusters, maximum distance between 2 prototypes. All these parameter values used in clustering and classification are decided based on the precision and recall value. In other words, until maximum F-measure is obtained, these parameters are changed.

## CHAPTER 5

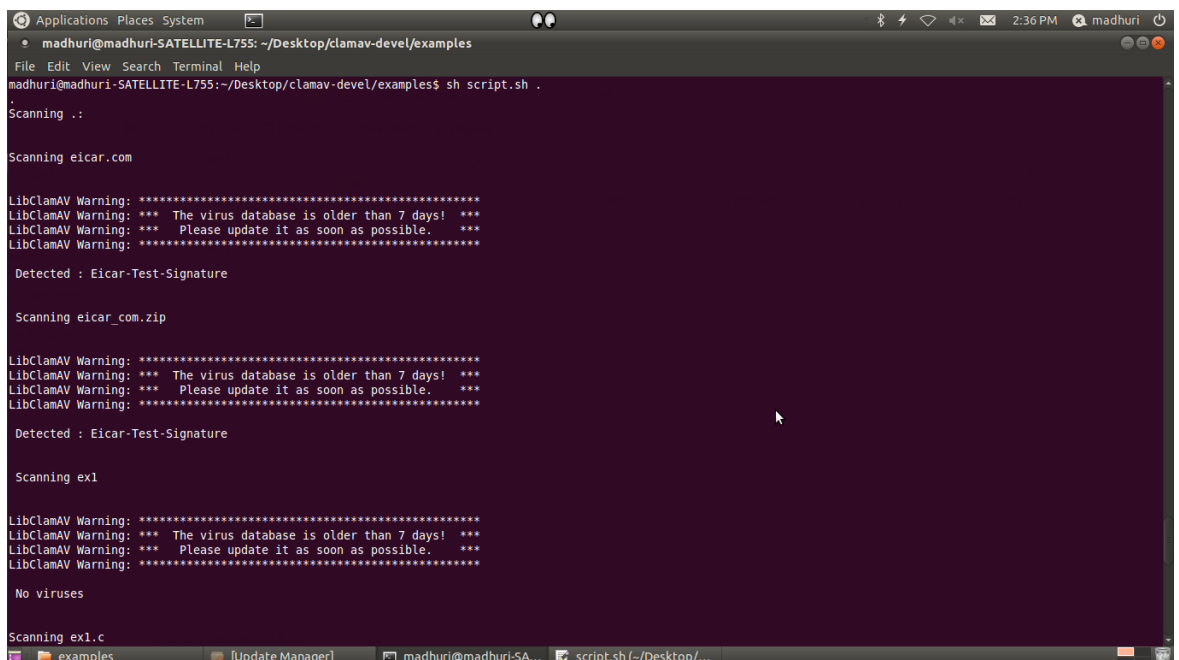
### SCREENSHOTS



A terminal window titled 'Applications Places System' with a menu bar 'File Edit View Search Terminal Help'. The prompt is 'madhuri@madhuri-SATELLITE-L755: ~/Desktop/clamav-devel/examples'. The command 'sh script.sh .' is entered. The output shows a scan of 'a.out' with no viruses detected. Then 'eicar.com' is scanned, detecting 'Eicar-Test-Signature'. Finally, 'eicar\_com.zip' is scanned, also detecting 'Eicar-Test-Signature'. The terminal has a dark purple background. The bottom status bar shows 'examples', '[Update Manager]', 'madhuri@madhuri-SA...', and 'script.sh (~/Desktop/...'.

```
madhuri@madhuri-SATELLITE-L755: ~/Desktop/clamav-devel/examples
File Edit View Search Terminal Help
madhuri@madhuri-SATELLITE-L755:~/Desktop/clamav-devel/examples$ sh script.sh .
Scanning .:
Scanning a.out
LibClamAV Warning: *****
LibClamAV Warning: *** The virus database is older than 7 days! ***
LibClamAV Warning: *** Please update it as soon as possible. ***
LibClamAV Warning: *****
No viruses
Scanning eicar.com
LibClamAV Warning: *****
LibClamAV Warning: *** The virus database is older than 7 days! ***
LibClamAV Warning: *** Please update it as soon as possible. ***
LibClamAV Warning: *****
Detected : Eicar-Test-Signature
Scanning eicar_com.zip
LibClamAV Warning: *****
LibClamAV Warning: *** The virus database is older than 7 days! ***
LibClamAV Warning: *** Please update it as soon as possible. ***
LibClamAV Warning: *****
Detected : Eicar-Test-Signature
Scanning ex1
```

Figure 5.1: Screenshot 1 of signature based detection



A terminal window similar to Figure 5.1, but the scan order is different. It starts with 'eicar.com', which detects 'Eicar-Test-Signature'. Then 'eicar\_com.zip' is scanned, also detecting 'Eicar-Test-Signature'. Then 'ex1' is scanned, with no viruses detected. Finally, 'ex1.c' is scanned, with no viruses detected. The terminal has a dark purple background. The bottom status bar shows 'examples', '[Update Manager]', 'madhuri@madhuri-SA...', and 'script.sh (~/Desktop/...'.

```
madhuri@madhuri-SATELLITE-L755: ~/Desktop/clamav-devel/examples
File Edit View Search Terminal Help
madhuri@madhuri-SATELLITE-L755:~/Desktop/clamav-devel/examples$ sh script.sh .
Scanning .:
Scanning eicar.com
LibClamAV Warning: *****
LibClamAV Warning: *** The virus database is older than 7 days! ***
LibClamAV Warning: *** Please update it as soon as possible. ***
LibClamAV Warning: *****
Detected : Eicar-Test-Signature
Scanning eicar_com.zip
LibClamAV Warning: *****
LibClamAV Warning: *** The virus database is older than 7 days! ***
LibClamAV Warning: *** Please update it as soon as possible. ***
LibClamAV Warning: *****
Detected : Eicar-Test-Signature
Scanning ex1
LibClamAV Warning: *****
LibClamAV Warning: *** The virus database is older than 7 days! ***
LibClamAV Warning: *** Please update it as soon as possible. ***
LibClamAV Warning: *****
No viruses
Scanning ex1.c
```

Figure 5.2: Screenshot 2 of signature based detection

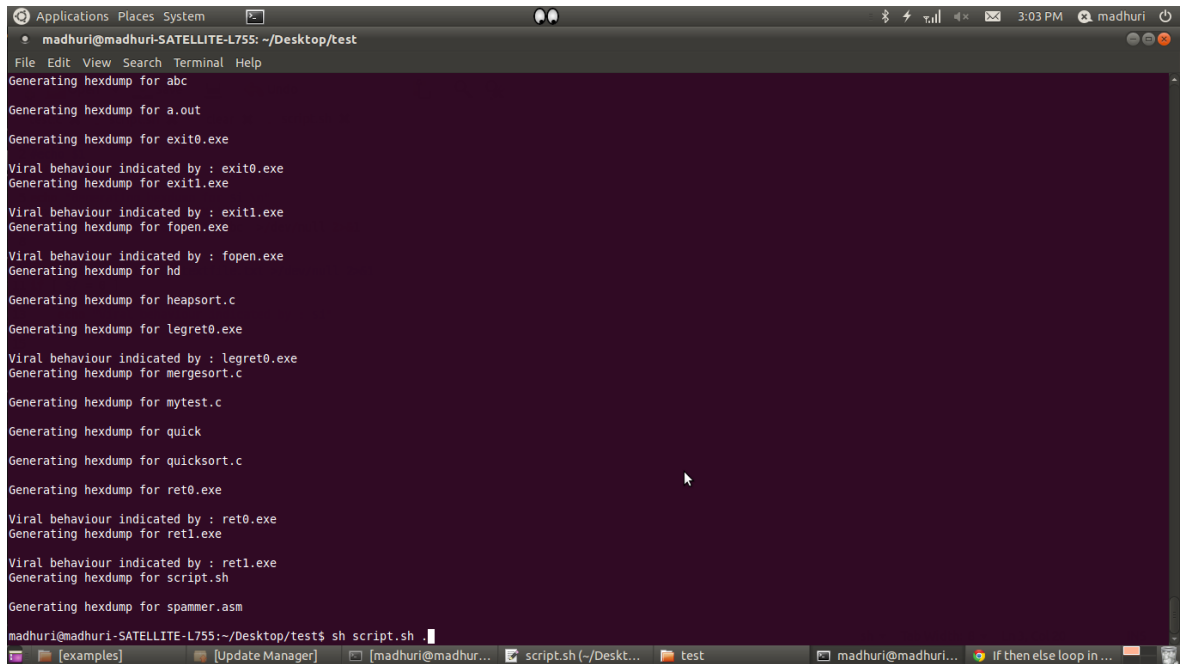


Figure 5.3: Screenshot of static heuristic scanning

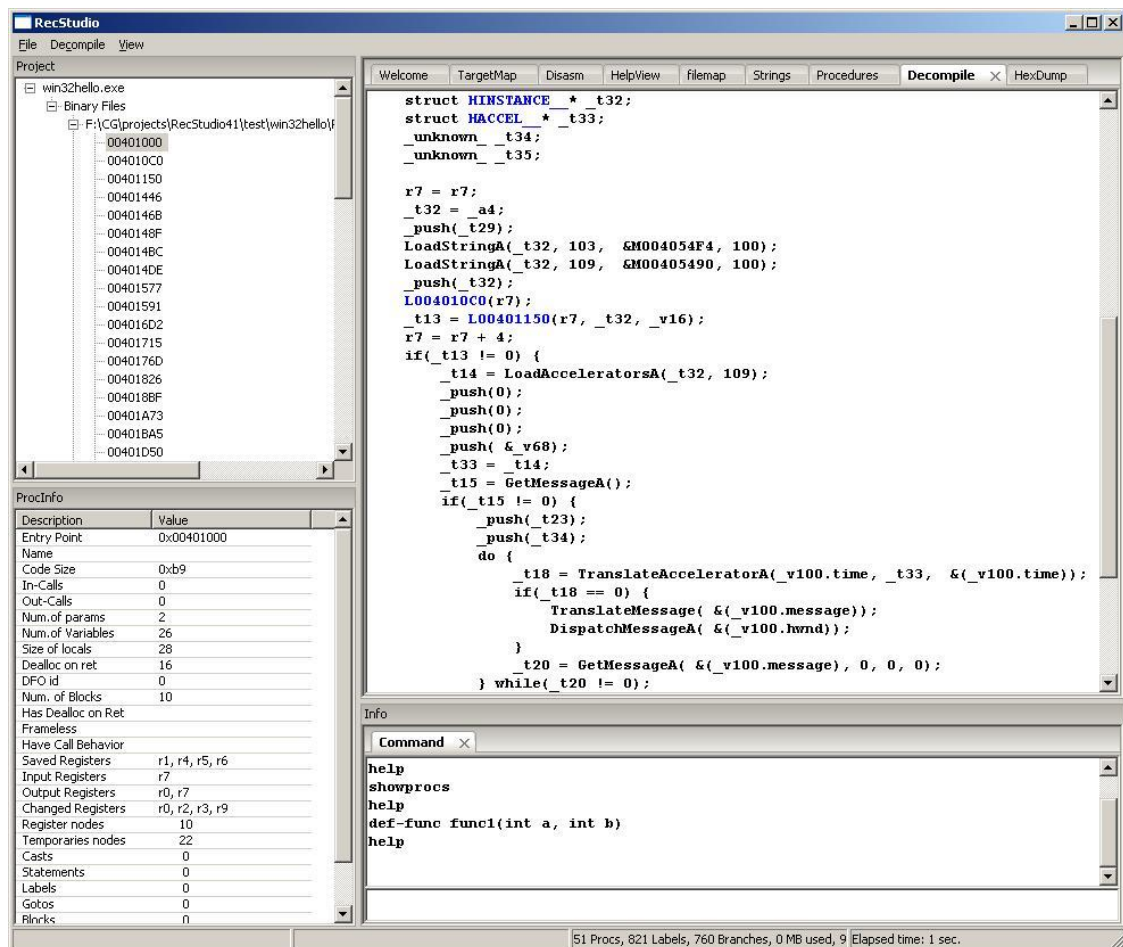


Figure 5.4: Screenshot of decompilation

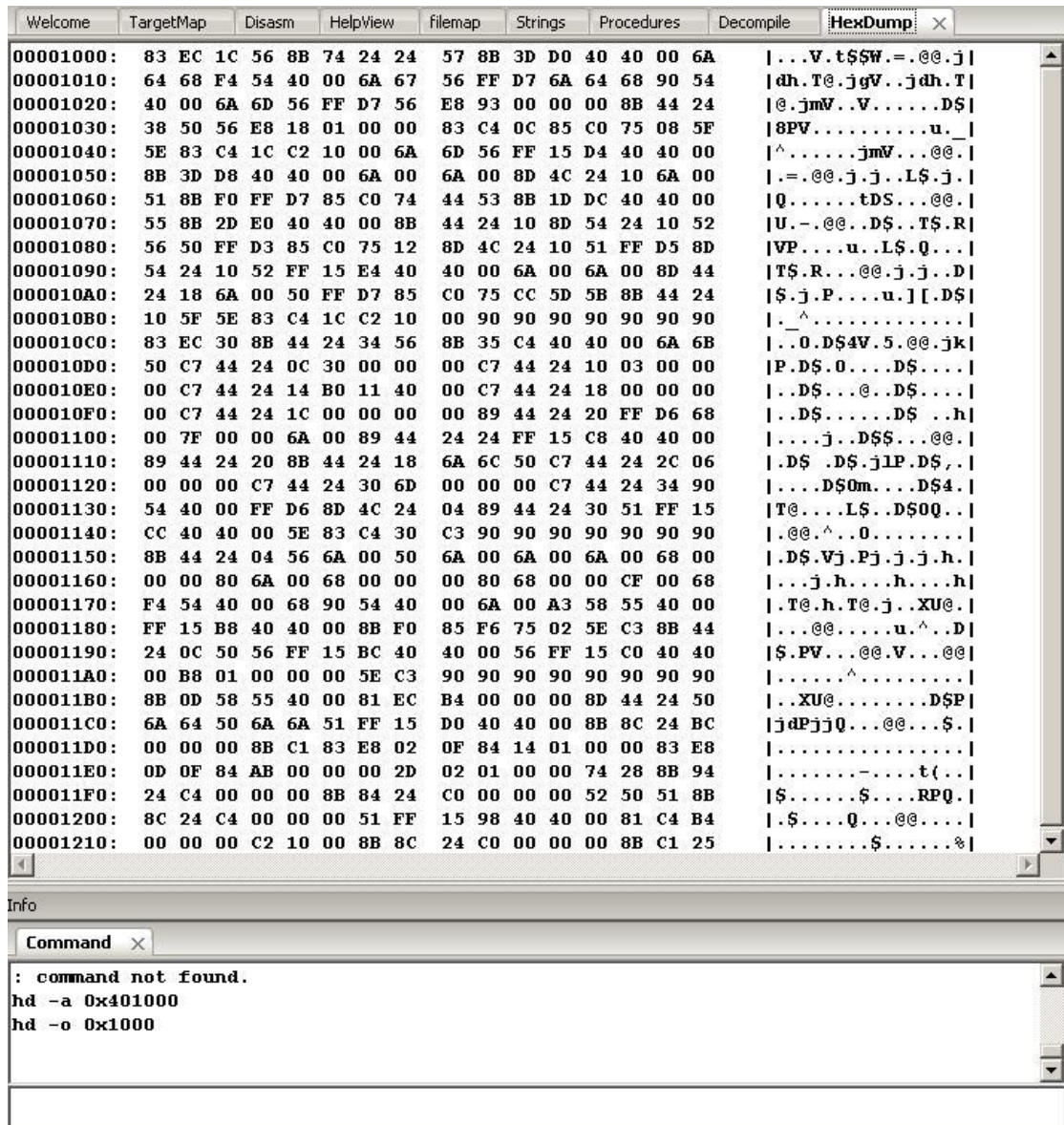


Figure 5.5: Screenshot of hexadecimal dump (Decompiler – RecStudioLinux Appendix )



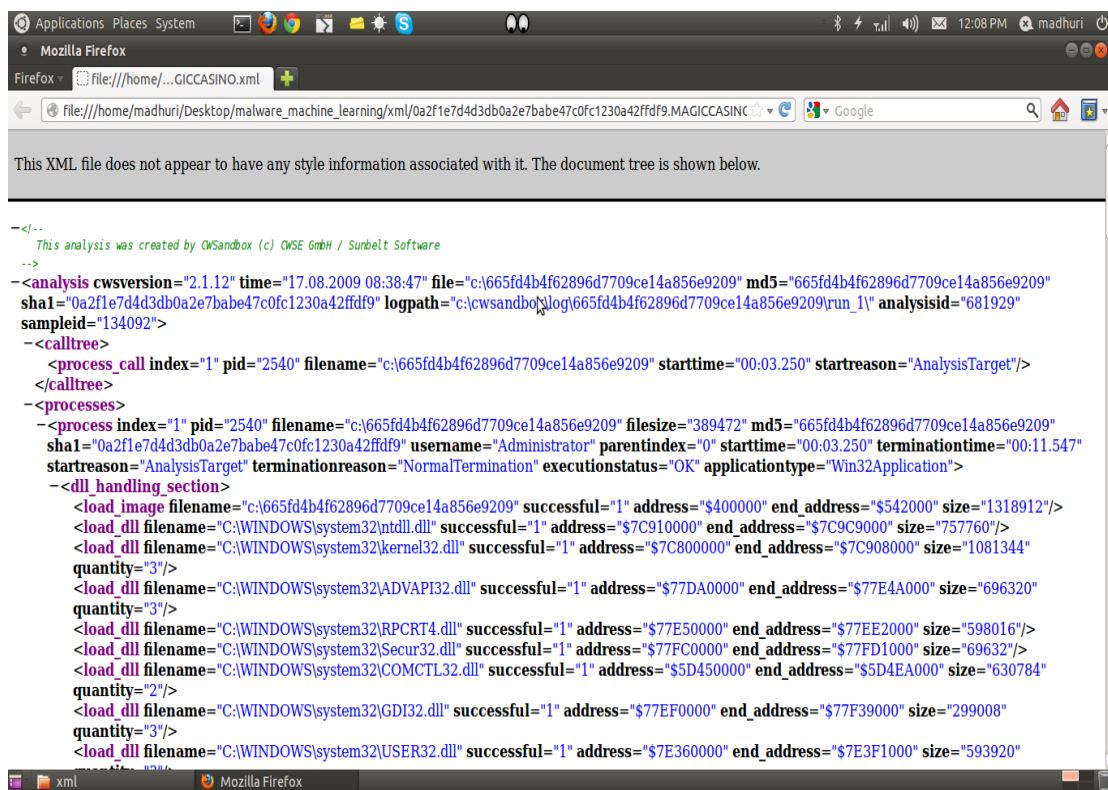


Figure 5.6: Screenshot of dynamic heuristic scanning – Sunbelt CWSandbox report

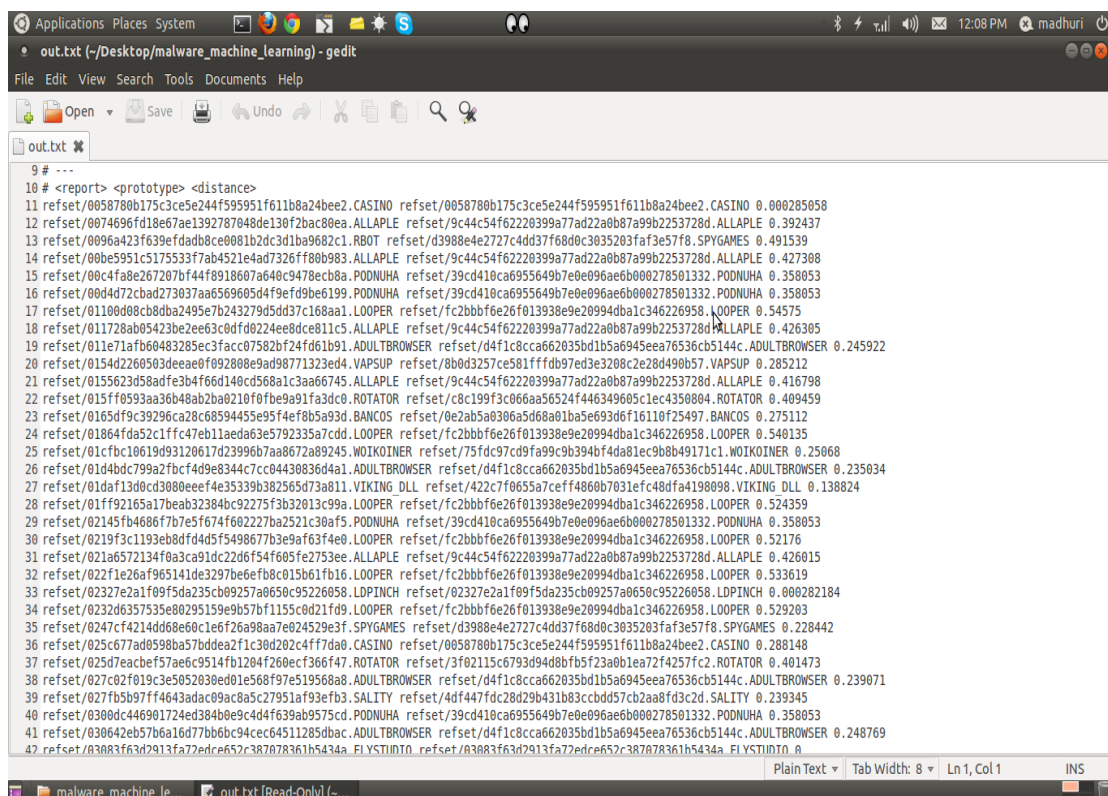


Figure 5.7: Screenshot of incremental analysis of malware behaviour

## CHAPTER 6

### CONCLUSION AND FUTURE WORK

Based on the results obtained, we conclude the following -

- Signature based detection fails to detect new viruses but is a high speed detection
- Static heuristic scanning helps achieve behavioural detection but results in false positives and is not highly robust
- Static heuristic scanning involves detailed static code analysis. However it is faster than dynamic scanning as it does not involve CPU emulation.
- Static heuristic scanning will require more complicated code analysis in case of oligomorphic viruses (viruses that encrypt themselves). Dynamic scanning will be more efficient but requires CPU emulation.
- Dynamic heuristic scanning takes care of obfuscated and encrypted viruses, however it results in false positives
- False positives may be reduced using Learning techniques. Case based reasoning is a good machine learning technique that can be used for malware detection. Clustering and classification may be used as complimentary paradigms rather than being used exclusively.
- Combining clustering and classification has the following advantages
  - Reduction of false positives
  - Better Precision in classification than Support Vector Machines (SVM)
- Future work –
  - Implementation for a larger suite of viruses
  - Parallelizing portions of the incremental algorithm on multi core platforms to achieve detection/classification at a faster rate.

## 6.1. WORK PLAN

Sl. No.	Estimated Deadline (End of the month)	Work	Progress
1	September, 2012	Design and Implementation of signature based detection.	Completed
2	November, 2012	Design and Implementation of static heuristic techniques for DOS viruses	Completed
3	February, 2013	Design and Implementation of static heuristic techniques for Kernel viruses, Process viruses and memory resident viruses.	Completed
4	March, 2013	Design and Implementation of dynamic heuristic scanning and learning techniques	Completed
5	April, 2013	Learning techniques and documentation	Completed

## REFERENCES

Xiao-bin Wang; Guang-yuan Yang, Yi-chao Li, Dan Liu (2008). "Review on the application of artificial intelligence in antivirus detection system," *Proc of 2008 IEEE Conference on Cybernetics and Intelligent Systems*, 506-509.

Xi Zhang, D. Saha, and Chen Hsiao-Hwa (2005). "Analysis of Virus and Anti-Virus Spreading Dynamics," *Prof of IEEE GLOBECOM*.

XianweiZeng, Zhijun Zhang, and Zhi Zhang (2005). "Heuristic skill of computer virus analysis based on virtual machine," *Proc. of Computer Applications and Software*, Vol. 22(9), 125-126.

Symantec Corporation, Understanding Heuristics (1997). "Symantec's Bloodhound Technology," *Symantec White Paper Series*.

Zhenhai Wang and Haifeng Wang (2006). "Study on Anti-Virus Engine Based on Heuristic Search of Polymorphic Virus Behaviour," *Proc. of Research and Exploration in Laboratory*, Vol. 25(9), 1089-1108.

Matthew G. Schultz, EleazarEskin, ErezZadok, and Salvatore I. Stolfo (2001). "Data Mining Methods for Detection of New Malicious Executables," *Proc. of the 2001 IEEE Symposium on Security and Privacy, Oakland*. CApp.38-49.

Yufeng Yang (2005). "The Network Virus Precaution System Based on DataMining," *Journal of Shaoguan University*, Vol. 26(12), 31-33.

Robert Moskovitch, Ido Gus, Shay Pluderman, DimaStopel, ChananGlezer et al (2007). "Detection of Unknown Computer Worms Activity Basedon Computer Behavior using Data Mining," *Proc. of the 2007 IEEE Symposium on Computational Intelligence in Security and Defense Applications (CISDA 2007)*, 169-177.

Lee J.S., Hsiang J., Tsang P.H (1997) "A Generic Virus Detection Agent on the Internet," *Proc. of the Thirtieth Hawaii International Conference on System Sciences, Wailea, HI*, 210-219.

T. Okamoto and Y. Ishida (1999). "A Distributed Approach to Computer Virus Detection and Neutralization by Autonomous Heterogeneous Agents," *Proc. of International Symposium on Autonomous Decentralized Systems (ISADS)*, 328-331.

Maoguang Wang, Zhaolin Yin, Helong Ding, and Xianzhong Zhang (2002). "The Application of Agent in Virus Detection," *Network & Computer Security*, 55-57.

Xiantai Gou, Weidong Jin, and Duo Zhao. (2006). "MULTI-AGENT SYSTEM FOR WORM DETECTION AND CONTAINMENT IN METROPOLITAN AREA NETWORKS," *JOURNAL OF ELECTRONICS (CHINA)*, Vol. 23(2), 259-265.

IBM (1996). "Combating computer viruses: IBM's new computer immune system," *IEEE Parallel and Distributed Technology*.

Harmer P., Williams P., Gunsch G., and Lamont G.B. (2002). "An Artificial Immune System Architecture for Computer Security Applications," *Proc. Of IEEE Transactions on Evolutionary Computation*, Vol. 6(3), 252-280.

Zhen Yu; Jianhui Ma; Xianbin Cao, and Xufa Wang (2004) "A Virus Detection Algorithm Based on Immune Associative Memory," Vol. 34(2), 246-252.

Kephart, J.O. (1995). "Biologically inspired defenses against computer viruses," *Proc. of International Joint Conference on Artificial Intelligence*, 985-986.

William Arnold and Gerald Tesauro (2000). "Automatically generated Win32 heuristic virus detection," *Proc. of the 2000 International Virus Bulletin Conference*, 51-60.

Fukumi M., Mitsukuwa Y., and Akamatsu N (2000). "A new rule generation method from neural networks formed using a genetic algorithm with virus infection," *Proc. of the International Joint Conference on Neural Networks, Como, Italy*, Vol. 3, 413-418.

Chen Guo; Jiarong Liang; and Meilian Liang (2005). "Method of Virus Detection Based on BP Neural Networks," Vol. 31(2), 152-156.

Boyun Zhang and Jianping Yin, Dingxing Zhang, Jingbo Hao, Shulin Wang (2007) "Computer viruses detection based on ensemble neural network," *Proc. Of Computer Engineering and Applications*, Vol. 43(13), , pp. 26-29.

Marmelstein R.E., Van Veldhuizen D.A., and Lamont G.B. (1998). "A Distributed Architecture for an Adaptive Computer Virus Immune System," *Proc. Of IEEE International Conference on Systems, Man, and Cybernetics*, Vol.4, 3838-3843.

White R (1998). "Open Problem in Computer Virus Research," *Virus Bulletin Conference, Munich Germany*.

Chamorro, E., Jianchao Han, Beheshti, M (2012). The Design and Implementation of an Antivirus Software Advising System," *Proc. Of Information Technology: New Generations (ITNG), 2012 Ninth International Conference*, vol., 612-617.

Symantec Corporation, (1997). "Understanding Virus Behavior under Windows NT", *Proc of Symantec White Paper Series*.

N.V.Narendra Kumar, R.K.Shyamasundar, George Sebastian and Saurav Yashaswee. "Algorithmic Detection of Malware via Semantic Signatures." In *European Institute for Computer Antivirus Research, EICAR Annual Conference Proceedings*, pages 49-73, 2011.

Mihai Christodorescu and Somesh Jha. 2003. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*(SSYM'03), Vol. 12. USENIX Association, Berkeley, CA, USA, 12-12.

Chieh-Jen Cheng; Chao-Ching Wang; Wei-Chun Ku; Tien-Fu Chen; Jinn-Shyan Wang, "A Scalable High-Performance Virus Detection Processor Against a Large Pattern Set for Embedded Network Security," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* , vol.20, no.5, pp.841,854, May 2012

## **APPENDIX I**

### **REC STUDIO**

REC Studio is an interactive decompiler.

It reads a Windows, Linux, Mac OS X or raw executable file, and attempts to produce a C-like representation of the code and data used to build the executable file. It has been designed to read files produced for many different targets, and it has been compiled on several host systems.

REC Studio 4 is a complete rewrite of the original REC decompiler. It uses more powerful analysis techniques such as partial Single Static Assignment (SSA), allows loading Mac OS X files and supports 32 and 64 bit binaries.



## **SUNBELT CWSANDBOX**

CWSandbox uses API hooking to dynamically analyze the malware. While the report won't be as contextually easy to understand as the reports provided by ThreatExpert, the CWSandbox report contains more of the (sometimes extraneous) modifications made by the malware. CWSandbox supports more filetypes than ThreatExpert, with new types like XML representation of reports constantly being added.

It provides a sandboxed environment to run malware binaries. In a sandboxed environment, each process is allocated its own memory. Thus CPU emulation can be done on a virtual environment without affecting the actual CPU.