SENRA Academic Publishers, British Columbia
Vol. 8, No. 1, pp. 2775-2782, February 2014
Online ISSN: 1920-3853; Print ISSN: 1715-9997

Canadian Journal of pure&applied sciences

DESIGN OF GENERIC ANTIVIRUS SYSTEM

Osaghae OE, *Egbokhare, FA and Chiemeke SC
Department of Computer Science, University of Benin, Benin City, Edo State, Nigeria

ABSTRACT

Antivirus software developers are advocating for sophisticated antivirus designs to implement their antivirus systems. However, the current antivirus systems heavily rely on updating of their malicious signature databases to detect malicious codes in executable programs. The problem with frequent update of malicious signatures databases is that it is not scalable; it cannot detect malicious code whose signature is not in the malicious signature database. Consequently, we designed a generic antivirus system that does not contain malicious database but rather, malicious codes are detected by the type of operating system functions used by the executable program. The proposed generic antivirus system uses deterministic finite automata, Naïve Bayes and Chi square techniques to detect malicious codes in executable programs. When the generic antivirus system is deployed to any operating system environment, malicious codes can be accurately detected in executable programs without a need to update its malicious signature database.

Keywords: Antivirus scanners; Malicious Software; Malicious Signatures.

INTRODUCTION

Antivirus (AV) scanners are used in an attempt to directly protect computer systems from damages. AV scanners detect a specific type of unauthorized activity in the form of malicious code, collectively known as malware. A recent study shows that 81% of all computer users have antivirus software installed on their computer (Feng, 2008). Malicious code is a computer program that modifies a system call or the functioning of a program without the consent of the user of the system. Malicious codes can be classified as virus, worm or a Trojan horse (Devara and Murali, 2012). A virus is a computer program that does not have the capability to replicate on their own, and rely on using other computer programs as a host in order to spread (Microsoft, 2004). A worm can be defined as malicious code which is either requires human intervention or not in order to propagates through a network. The release of worms on computer networks has cost billions of dollars in wasted time and resources. Trojan horse is a non-replicated malicious code designed to cause damage to computer systems, by masquerading as benign programs. A Trojan horse is also regarded as a computer program that appears to have a useful function, but also has a hidden and potentially malicious function. A benign program is an executable program that does not contain any malicious code (Harley and Lee, 2009; Greensmith and Aickelin, 2005; Tikkanen, 2010).

The most common approach developed in anti-virus software products and tools to identify the viruses and malwares is signature-based scanning. It makes use of small strings, named as signatures, which are the results of manual analysis of viral codes. A signature must only be a sign of a specific virus and not the other viruses and normal programs. Accordingly, a virus would be discovered, if the virus related signatures were found Babak *et al*. (2011).

Antivirus definitions are databases that contain information used to identify viruses. Antivirus scanning engines are designed to identify specific viruses using the aforementioned definitions and by recognizing characterized behaviour. Antivirus software vendors release a new virus definition (databases) for their software products when they find new viruses. These vendor-specific database definitions are used by antivirus software to identify known viruses and/or virus-like behaviour. When information about a specific virus is included in a virus definition, it is said to be a known virus NetApp (2006).

Computer malwares can be classified according to their infection mechanism. The mechanism can be in the form of Encryption, oligomorphism, polymorphism and metamorphism. Encrypted virus changes its body binary code with some encryption algorithms to hide it from simple view and make it more difficult to analyze and detect. Oligomorphic virus substitute decryptor code in new offspring, which makes the detection process more difficult for signature based technique. A polymorphic virus is a malicious code that when it decides to infect a new victim program, it modifies some pieces of its body to look dissimilar. Polymorphic virus has capability to create infinite new decryptors. Metamorphic virus mutates all of its body and it also changes the code of decryption loop (Babak *et al*., 2011).

*Corresponding author email: fegbokhare@yahoo.com

The first generation of antivirus products was purely based on signature detection technique. The second generation made attempt to identify and stop network worms based on packet signatures. It also has the ability to disinfect and restore the Operating System from spyware or Trojan backdoor infection. The third generation was developed to effectively block zero-day malware proactively without any dependency on viral signatures. It uses behavioural analysis and behavioural blocking. Behaviour analysis is process to intercept API calls made by an executable program to determine if these API calls are used for malicious intent or not. To prevent malicious attacks generically, it is cost-effective to use behavioural blocking technique to restrict the actions that authorized executable programs can perform in the system when it is noticed that it contains malicious API calls Kumar and Spafford (1992).

The specific objectives of this study are to review the extent of improvements made to the antivirus systems and design a generic antivirus system that detects malicious codes in executable program based on how its uses operating system functions.

Kumar and Spafford (1992) described a virus detection tool called a generic virus scanner. This tool is completely general and is structured in such a way that it can easily be augmented to recognize viruses across different system platforms with varied file types. The implementation defines virus features common to all scannable viruses. The approach used to develop this tool is easy to understand. By combining string sets, it is believed that the coverage may be obtained in a manner superior to most commercial scanners currently available. The tool uses a pattern matching technique to identify malicious signatures in an executable program (Kumar and Spafford, 1992) and this can be evaded by sophisticated malware codes.

Roberto *et al.* (2004) proposed a WHIPS prototype also known as a Reference Monitor (RM) for the detection and prevention of Windows dangerous system calls invocation. WHIPS was designed and implemented to stop common exploits that use the buffer overflow technique to carry out privilege escalation on a system. If a malicious user wants to execute a shell in a context of the exploited service, WHIPS will prevent the attack by stopping the execution of the dangerous system call that invokes the shell. WHIPS is implemented as a kernel driver, also called kernel module, using the undocumented structure of the Windows kernel and the routines typically employed for driver development. The problem with WHIPS is that it was considered to be more efficient if it were implemented directly into the Windows kernel, instead of as a kernel driver Roberto *et al.* (2004).

Antivirus scanners of first generation employed non-complicated techniques in order to find known computer viruses. These set of scanners typically looked for certain patterns or sequences of bytes called string signatures. Antivirus scanners of the second generation was introduced when the earlier scanners lost their efficiency by using simple pattern scanning techniques to detect newer and more complicated viruses appropriately. Then the second generation of scanners introduce almost exact recognition that caused the antivirus scanners became more trustable. Antivirus scanners of the third generation use virus specific detection algorithm. This type of detection algorithm denotes any special method that is specifically designed for a given particular virus. This technique may bring about many problems such as portability of the scanner on different platforms and stability of code. To overcome these problems, virus-scanning languages have been developed that in scanned objects are allowed. Antivirus scanners of the fourth generation simulates the computer central processor, main memory, storage resources and some necessary functions of operating system by a virtual machine to run the malware virtually and investigate its behaviour and performance. The malicious code does not execute on actual machine and it is controlled by the virtual machine precisely, therefore there is no risk for unintentionally propagation of malware. These set of antivirus scanners can detect encrypted, polymorphic, metamorphic and oligomorphic viruses (Babak *et al*., 2011).

A packer is a software tool that can modify and compress an executable file by encrypting and changing its form from its original format. The final result is a modified executable which, when executed, does exactly the same thing as the original code, but from the outside has a completely different form and therefore evades signature-based unless either the engine has the specific unpacking algorithm or it is able to unpack it generically (Pedro *et al*., 2012; Guo *et al*., 2012).

The most important piece in any antivirus infrastructure is the virus definition file. Antivirus product clients keep their protection current by regularly updating virus definition files. These files contain the signatures of all the known viruses and are used by the scan engine. When a new virus comes out, the definition files need to be updated so that client software can detect the new virus. The definition files also give the client instructions on how to clean viruses from a file. Updating virus definition files quickly and efficiently is crucial in any business, especially in a virus situation. In a well-designed antivirus architecture, the clients will automatically update virus definition files on a regular basis (Speice, 2003; Morton, 2010).

MATERIALS AND METHODS

METHODOLOGY

In this section, we describe the design of the proposed antivirus system shown in figures 1, 2 and 3. One way to begin the design of any program is to describe the behaviour of the program by a Conceptualized Diagram. In an operating system, an executable program makes a set of system function calls $S_1, S_2, S_3 \ldots S_n$. The system functions take various numbers of parameters $S_1(P_1, P_2, P_3, \ldots, P_{c1})$, $S_2(P_1, P_2, P_3, \ldots P_{c2})$, $S_3(P_1, P_2, P_3, \ldots P_{c3})$, …, $S_n(P_1, P_2, P_3, \ldots, P_{ct})$. Where n is the number of system function calls made by the executable program, and *c1, c2, c3* and *ct* are the number of parameters used by the system functions. The purpose of this design is to define how the proposed antivirus system will detect the set of malicious system function calls made by an executable program. Figure 1 shows the behaviour of the feature extraction phase of Detection system. An executable program can contain executable *statement_1(S_1), statement_2(S_2), statements_3(S_3),…, and statement_n(S_n)*. Each of these executable statements can either be code or data. If it is code, it can be converted to

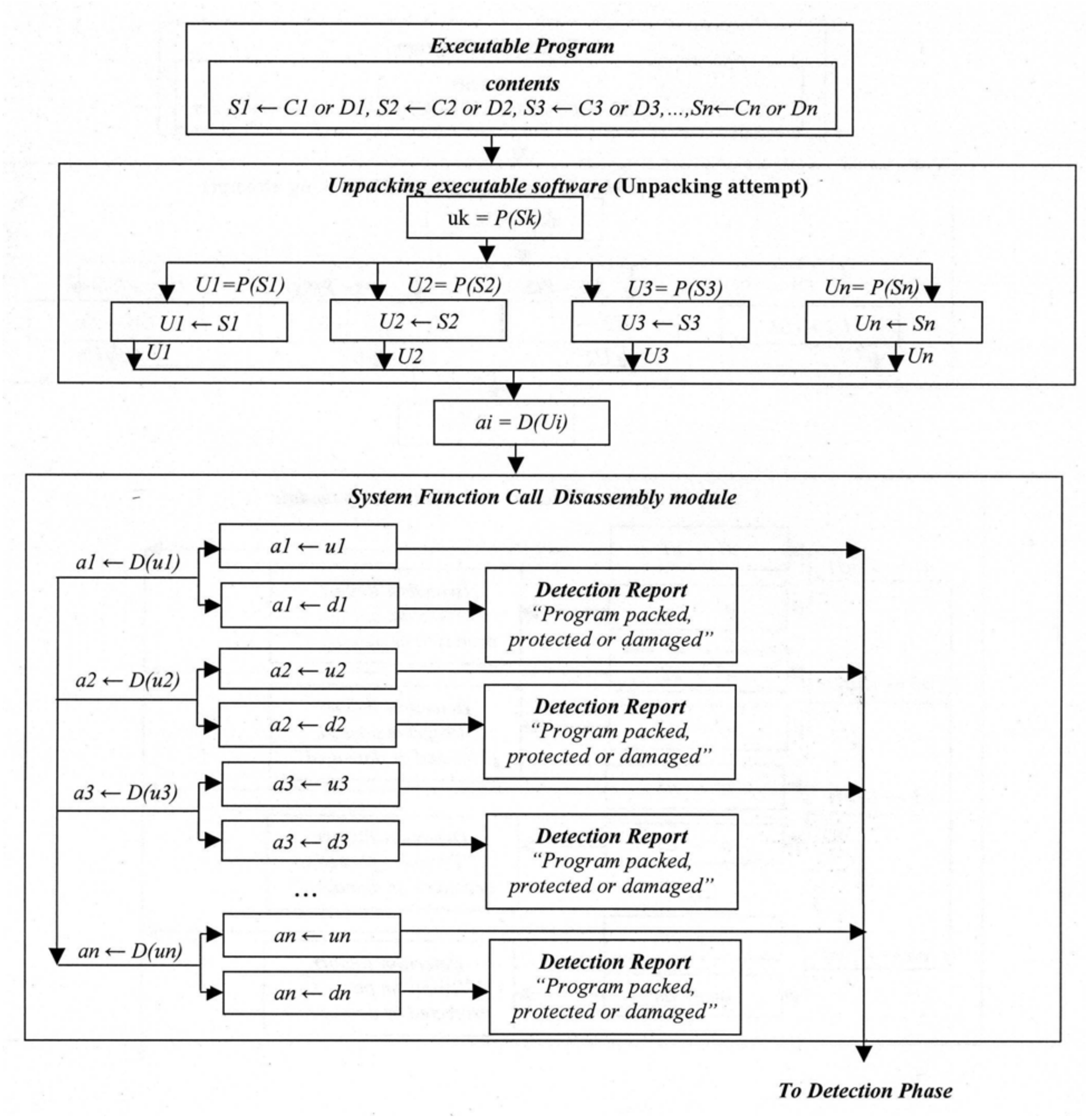

Fig. 1. Feature Extraction Phase of Detection System.

assembly code using a disassembler but if it is data then, it has been packed. The content of the program is sent to the Unpacking section which will attempt to convert an identified executable data statement back to its equivalent executable code. The unpacking executable software section uses a procedure called P(Sk). *P(Sk)* procedure takes one argument at a time called *statement (Sk)* and attempts to covert it to its equivalent executable code using a procedure called *unpacked (Uk)*. When the unpacking section has finished its function, it will pass the processed content of the executable program to the disassembly section. The System Call Disassembly module takes the responsibility of converting the sequences of executable statements to equivalent assembly code statements. This module has a procedure called *D(ui)* which takes unpacked executable code *ui* as argument and produces an equivalent assembly code statement *ai*. It disassembles each of the unpacked executable statement and produces their assembly code equivalent *ai* in the form of $a1 \leftarrow D(u1)$, $a2 \leftarrow D(u2)$, $a3 \leftarrow D(u3)$,…, $an \leftarrow D(un)$. When unpacked executable code *ui* is successfully converted to assembly code equivalent *ai*, the virus detector expresses it in the form of $a1 \leftarrow u1$, $a2 \leftarrow u2$, $a3 \leftarrow u3$,…, $an \leftarrow un$.

When the System Function Call Disassembly module is unable to convert the unpacking executable code *ui* into its equivalent assembly code, then there is a problem. The problem is the Unpacking Executable software module was not able to unpack the program content s*1, s2, s3,…, sn*. The implication of this is that the virus detector expresses the program in the form of $a1 \leftarrow d1$, $a2 \leftarrow d2$, $a3 \leftarrow d3$,…, $an \leftarrow dn$. The found components *d1*, *d2*, *d3*,…, *dn* were not successfully convert to their equivalent unpacked executable codes by the Unpacking Executable Software used. As it is done in conventional antivirus software systems, when a packed executable program cannot be unpacked by series of unpacking software tools, the program is reported as a malicious infected program.

In the Detection Phase, the arrays to store the total number of malicious attributes found, *self-modification*, *self-referential*, *self-replication*, malicious system functions for worms and Trojan are initialized to null. The array names for malicious attributes found are general malicious category, *self-modification*, *self-referential*, *self-replication*, malicious system functions for Trojan and worm are mal, *VDSM, VDSR1, VDSR2, MST* and *MSW* respectively.

When control reaches the Detection phase, these arrays are initialized to null and control is passed to the procedure named *SearchSystemFunc(ak)*. The *SearchSystemFunc(ak)* is a procedure which takes one argument *ak*; assembly code derived by the disassembly module. The responsibility of the *SearchSystemFunc(ak)* is to search and collect the set of system functions in the assembly code presented. When a system function is found, it is stored in a variable called *sfk*. The variable *sfk* is passed as an argument to another procedure called *MalSystemFunction(sfk)*.

The *MalSystemFunction(sfk)* is responsible for identifying the set of malicious system functions by interacting with the set of definition given in the system call behaviour storage. When the procedure finds a malicious system function, it is stored in a variable called *mfk*. As soon as the procedure *MalSystemFunction(sfk)* finds a malicious system function, it is added to the array *mal.* This addition is cumulative until all the malicious system functions are collected. When the array *mal* is empty after the *MalSystemFunction(sfk)* has been called, the detector declares the executable program being examined as benign. When the array *mal* is not empty, the set of malicious system function collected are sent to the virus identification module.

The system call intelligent section uses the multinomial technique of Naïve Bayes classifier to extract the virus attributes (*self-modification*, *self-referential* and *self-replication* attributes) from the malicious attributes. Let C denote the set of class labels, that is:

$$C = \{C_1, C_2, C_3\} \quad (1).$$

Equation 1 is the set of malicious classes and the total number of classes is 3. Let GSF denote the Groups of System Functions(GSF), where

$$GSF= \{ GSF_1, GSF_2, GSF_3\} \quad (2)$$

Equation 2 is the set of GSF that can be got from the malicious classes.

In a virus identification section, when malicious system functions for the definitions for *self-modification (DSM)*, *self-referential (DSR1)* and *self-modification (DSR2)* are found, they are added to their various arrays *VDSM*, *VDSR1* and *VDSR2* respectively. The virus identification section has six procedures they are I*nVSM(DSM), OutVSM(DSM), InVSR1(DSR1), OutVSR1(DSR1), InVSR2(DSR2) and OutVSR2(DSR2).*

The procedure *InVSM(DSM)* takes one argument DSM and is responsible for accumulating the set of self-modification system functions found in a program. The procedure *OutVSM(DSM)* takes one argument DSM and is responsible for notifying the procedure InVSM(DSM) that the all identified self-modification system functions used by the executable program has been found and collected. Another procedure called *InVSR1(DSR1)* takes one argument DSR1 and is responsible for accumulating the set of self-referential system functions found in a program. The procedure *OutVSR1(DSR1)* takes one argument DSR1 and is responsible for notifying the procedure InVSR1(DSR1) that all identified self-

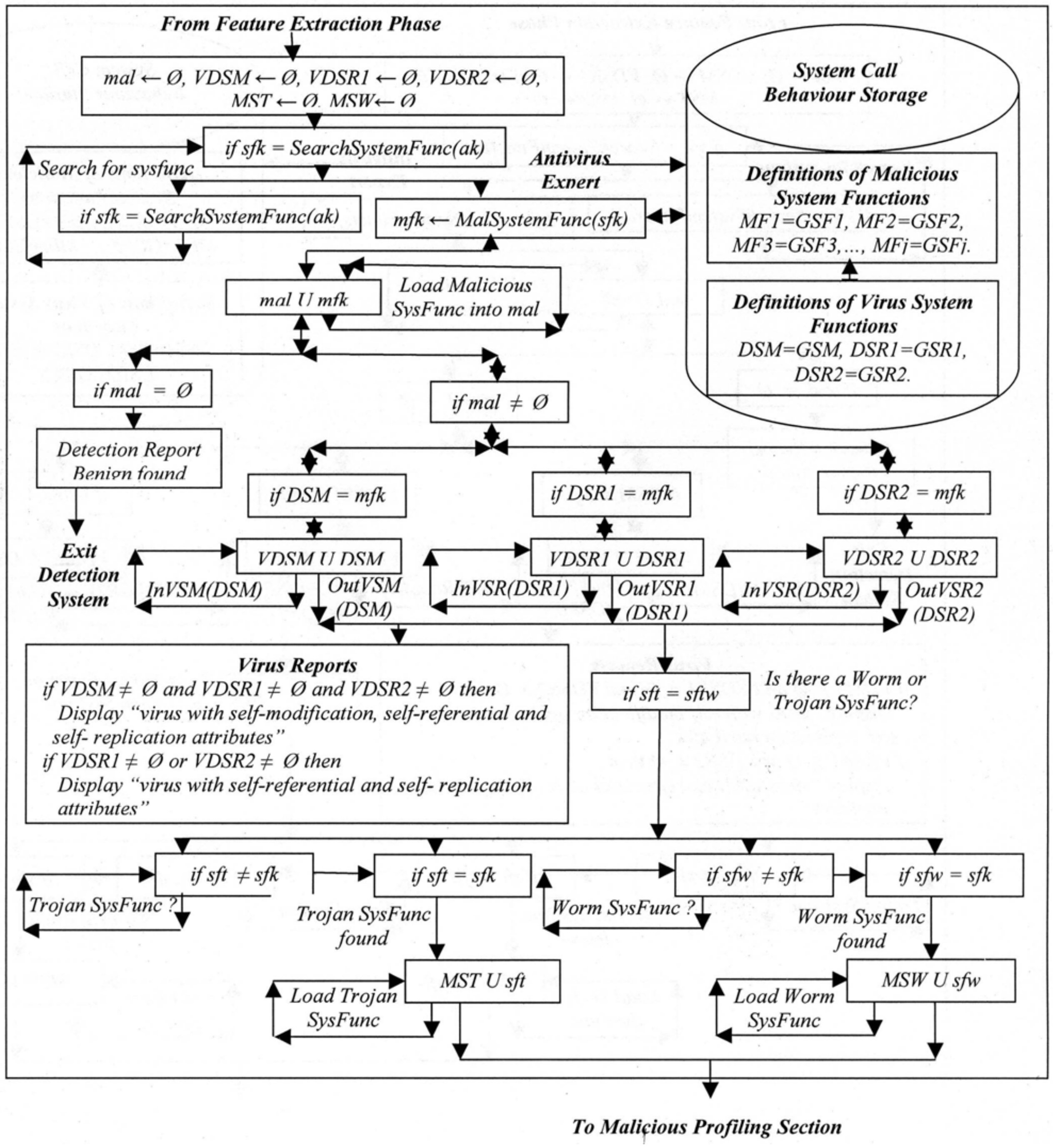
From Feature Extraction Phase
mal ← Ø, VDSM ← Ø, VDSR1 ← Ø, VDSR2 ← Ø, MST ← Ø, MSW ← Ø
if sfk = SearchSystemFunc(ak)
Search for sysfunc
if sfk = SearchSystemFunc(ak)
mfk ← MalSystemFunc(sfk)
Antivirus Expert
System Call Behaviour Storage
Definitions of Malicious System Functions
MF1=GSF1, MF2=GSF2, MF3=GSF3,..., MFj=GSFj.
Definitions of Virus System Functions
DSM=GSM, DSR1=GSR1, DSR2=GSR2.
mal U mfk
Load Malicious SysFunc into mal
if mal = Ø
if mal ≠ Ø
Detection Report Benign found
Exit Detection System
if DSM = mfk
if DSR1 = mfk
if DSR2 = mfk
VDSM U DSM
VDSR1 U DSR1
VDSR2 U DSR2
InVSM(DSM)
OutVSM (DSM)
InVSR(DSR1)
OutVSR1 (DSR1)
InVSR(DSR2)
OutVSR2 (DSR2)
Virus Reports
if VDSM ≠ Ø and VDSR1 ≠ Ø and VDSR2 ≠ Ø then
Display "virus with self-modification, self-referential and self- replication attributes"
if VDSR1 ≠ Ø or VDSR2 ≠ Ø then
Display "virus with self-referential and self- replication attributes"
if sft = sftw
Is there a Worm or Trojan SysFunc?
if sft ≠ sfk
Trojan SysFunc ?
if sft = sfk
Trojan SysFunc found
if sfw ≠ sfk
Worm SysFunc ?
if sfw = sfk
Worm SysFunc found
MST U sft
Load Trojan SysFunc
MSW U sfw
Load Worm SysFunc
To Malicious Profiling Section

Fig. 2. Detection Phase of Detection System.

referential system functions used by the executable program has been found and collected.

Another procedure called *InVSR2(DSR2)* take one argument *DSR2* and is responsible for accumulating the set of self-replication system functions found in a program. The procedure *OutVSR2(DSR2)* take one argument *DSR2* and is responsible for notifying the procedure *InVSR2(DSR2)* that all identified self-replication system functions used by the executable program has been found and collected. When the virus identification section has finished examining the executable program for the three attributes of a virus, it will send the detection results to a virus report section. In the virus report section, when the arrays for *self-modification(VDSM)*, *self-referential(VDSR1)* and *self-replication(VDSR2)* attributes are not empty, then the following comparisons are carried out. When an executable program contains *self-modification*, *self-referential* and *self-replication* system function attributes, then it is a virus program with these three attributes. After the virus detector has finished examining the executable program for evidence of possible virus infection, then the set of system functions used by the program are passed to the separator module.

In the separator module, there is an attempt to accumulate the set of system functions used by Trojan and the ones not used by worm. This will enable the virus detector to examine the executable program for Trojan and worm behaviour using chi square technique.

The detector check if the system function *sfk* belongs to the set of system functions *sftw* commonly used by Trojan *sft* and the ones not worm *sfw*. The total number of system functions used by Trojan and the ones not used by worm are store in variables *MST* and *MSW* respectively. After the separator module has finished examine the system functions of an executable program, the set of identified system functions for Trojan and the ones not used by a worm are passed to the Malicious Profiling section of the virus detector. The model for Chi square computation that the Malicious Profiling section will used to detect Trojan horses and worms, are presented in the form of a pseudocode below:

```
Begin
1) Define $P_i = (P_1, P_2)$ to be the set of profiles of samples in the malicious attribute where Trojan and worm system function has been identified from the classes $C_1$,(classes for Trojan) and $C_2$(classes for worm).
   Define $T = (T_1, T_2, T_3, ..., T_n)$ to be tested samples of a generalized malicious attributes belonging to the classes $C_1$, and $C_2$.
   Define Tr to be the set of system function calls used to identify a Trojan program.
   Define Wo to be the set of system function used to identify a worm program.
2) Get the program system function calls which belong to Tand W extracted by the separator section.
3) Compute chi-square for each attribute class for
   i = 1 to 2 do
      $X_i^2 = \frac{(A_i - V_i)^2}{V_i}$
   $A_i$ is the number of observed malicious attribute and i is 2; trojan and worm.
   $V_i$ is number of expected worm or Trojan attribute is suppose to have.
4) Compute degree of membership $\sigma_i = \sum X_i^2$.
5) Compute the degree of freedom = number_ of_ attributes – 1 Degree of freedom = 2 – 1=1
6) Compute threshold value $\sigma_{1,}$ where the null hypothesis judgment would be based upon, by reading the degree of freedom from probability level of $d$ from the Chi square table.
```

A significant level of 0.z based on the degree of freedom, will be selected. This means that z% of the time, X^2 is expected to be less than or equal to σ_1.
$X^2_{.z} \leq \sigma_1$.

```
7) Compute the classification strategy:

If
   $\exists\ i,\ 1 \leq i \leq 2,\ \sigma_i \geq \sigma_1$
      $\Rightarrow T \in Tr_i$.
Otherwise,
   $\forall\ i,\ 1 \leq i \leq 2,\ \sigma_i < \sigma_1$
      $\Rightarrow T \in Wo_i$.
```

The behaviour of Malicious Profiling section is displayed in figure 3. The purpose is to use the set of system functions passed from the separator section to identify possible Trojan and worm system functions of an executable program. By the time Malicious Profiling section receives the sets of system functions stored in *MST* and *MSW*, it attempts to identify Trojan and worm by using Chi square technique. Computation of a Chi square terms for Trojan is *CT ← MST – CTi* and the one for worm is *CW ← MSW – CWi* are used. *MST* and *MSW* are the observed total number of system functions for Trojan and worm, extracted by the separator section. *CTi and CWi* are the expected total number of system functions used by Trojan and the ones not used by worms. *CTi and CWi* are stored as definitions in the Malicious Profiling storage. The malicious profiling section is expected to interact with these definitions during its computation.

The final Chi square computation is *CM←(CT*CT)/CWi + (CW)*(CW)/CTi,* which determines the degree of membership. Next, the degree of freedom is computed as *Degree of Freedom←number of attributes (2) – 1*. Since we are concerned with two attributes namely, Trojan and worm, then the computation for degree of freedom is 1. The next stage is to compute the threshold value *TV* where the null hypothesis judgment would be based upon. The threshold value *(TV)* is got from the Chi square table by reading degree of membership against the degree of freedom. To identify Trojan and worm, the following comparisons are made. When an executable program contains malicious system functions and CM < TV, then it has Trojan codes. Again, when an executable program contains malicious system functions and CM ≥ TV, then it has Worm codes. Then, when an executable program contains malicious system functions and does not have the conditions CM < TV and CM ≥ TV, then it is an unknown malicious codes. When *mal* = *Ø* then the executable program being examine is benign. The final detection results are sent to the detection report section. The

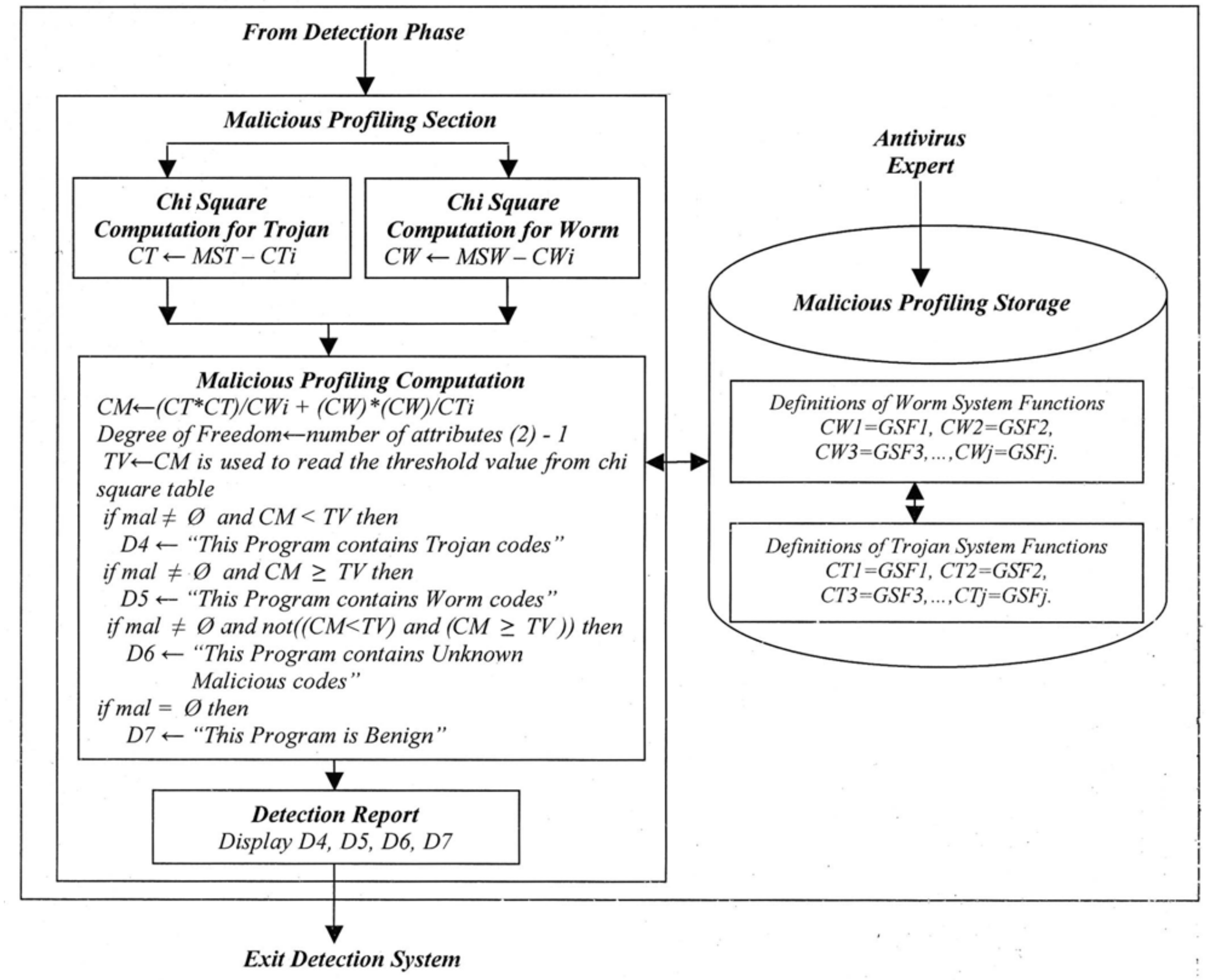
From Detection Phase
Malicious Profiling Section
Chi Square Computation for Trojan
CT ← MST – CTi
Chi Square Computation for Worm
CW ← MSW – CWi
Malicious Profiling Computation
CM←(CT*CT)/CWi + (CW)*(CW)/CTi
Degree of Freedom←number of attributes (2) - 1
TV←CM is used to read the threshold value from chi square table
if mal ≠ Ø and CM < TV then
D4 ← "This Program contains Trojan codes"
if mal ≠ Ø and CM ≥ TV then
D5 ← "This Program contains Worm codes"
if mal ≠ Ø and not((CM<TV) and (CM ≥ TV)) then
D6 ← "This Program contains Unknown Malicious codes"
if mal = Ø then
D7 ← "This Program is Benign"
Detection Report
Display D4, D5, D6, D7
Antivirus Expert
Malicious Profiling Storage
Definitions of Worm System Functions
CW1=GSF1, CW2=GSF2, CW3=GSF3,...,CWj=GSFj.
Definitions of Trojan System Functions
CT1=GSF1, CT2=GSF2, CT3=GSF3,...,CTj=GSFj.
Exit Detection System

Fig. 3. Malicious Profiling Section.

Malicious Profiling storage keeps the set of definitions for the expected worm and Trojan system functions. These definitions are determined by the antivirus expert who knows the exact number of system functions that make up the expected number of sets of worm and Trojan functions. The definition for worm system functions are *CW1=GSF1, CW2=GSF2, CW3=GSF3,...,CWj=GSFj. CWj* is the total number of Trojan system functions extracted from the Group of system functions present in the operating system where the program is running. Another Definition for system functions not used by worm are *CT1=GSF1, CT2=GSF2, CT3=GSF3,...,CTj=GSFj. CTj* is the total number of system functions not used by Trojan, and they are extracted from the Group of system functions in the operating system the program is running on. The detection results got from the Malicious profiling section is sent to detection report section.

The detection report section displays the Trojan, worm, unknown malicious codes and benign detection results got from the executable program. After the detection results have been reported by the detection report section, the virus detector exits.

CONCLUSION

In this paper, we designed a generic antivirus system that makes use of operating system functions rather than updating its malicious signature database. For an executable program to operate in the computer system, it must make use of the operating system functions also known as the system functions. As surprising as it seems, the operating system cannot differentiate between a set of system functions made by a benign program from the ones made by the malicious program. This paper has shown the design of a generic antivirus system which makes use of deterministic finite state automata, Naïve Bayes and Chi square to accurately detect malicious codes from executable programs. The detection of malicious codes is done in the absence of malicious signature database component. The design of a generic antivirus system proposed in this paper will be highly appreciated by antivirus developers. The appreciation will be in terms of *reduction of human involvement*, *more scalable antivirus design* and *elimination of malicious signatures database from antivirus design*. There is reduction of running cost because fewer antivirus experts are required and there is no need to perform malicious signatures extractions from each malicious program. The antivirus system design will be more scalable because a generic signature is defined than having a numerous unique malicious signature definitions. The operating system functions are used to detect malicious code in executable programs rather than using malicious signature database. Our future research direction on design of generic antivirus system is to attempt to deploy its design to the Windows operating system functions. We are also going to develop algorithms for the generic and proposed Windows operating system designs. We shall also attempt to implement the proposed Windows operating antivirus design in C programming language and then test live malicious programs antivirus system to determine its efficiency.

REFERENCES

Babak, BR., Maslin, M. and Suhaimi, I. 2011. Evolution of Computer Virus Concealment and Anti-Virus Techniques: A Short Survey. International Journal of Computer Science. 8(1):113-121.

Devara, V. and Murali, K. 2012. Network Based Anti-virus Technology for Real-time Scanning. International Journal of Computer Science Issues. 9(4):304-310.

Feng, X. 2008. Attacking Antivirus. Nevis Labs, Nevis Networks, Inc.

Greensmith, J. and Aickelin, U. 2005. Firewalls, Intrusion Detection Systems and Anti-Virus Scanners. School of Computer Science and Information Technology, University of Nottingham, Jubilee Campus, Nottingham, UK.

Guo. F., Ferrie, P. and Chiueh, T. 2012. A Study of the Packer Problem and Its Solutions. (Retrieved from www.ecsl.cs.sunysb.edu/tr/ TR237.pdf).

Harley, D. and Lee, A. 2009. Heuristic Analysis: Detecting Unknown Viruses. ESET Corporation, Bratislava, Slovak Republic.

Kumar, S. and Spafford E. H. 1992. A Generic Virus Scanner in C++. The COAST Project, Department of Computer of Computer Sciences, Purdue University, West Lafayette.

Microsoft. 2004. The Antivirus: Defense-in-Depth Guide. Microsoft Corporation.

Morton, C. 2010. Bypassing Malware Defenses. The SANS Institute.

NetApp. 2006. Antivirus Scanning Best Practices Guide. Network Appliance Inc., (Retrieved from www.netapp.com on 15/10/2012).

Pedro, B., Inaki, U., Luis, C. and Josu, F. 2012. From Traditional Antivirus to Collective Intelligence: Panda's Technology Evolution. Panda Research, (Retrieved from research.pandasecurity.com on 15/10/2012).

Roberto, B., Emanuele, G. and Luigi, VM. 2004. A Host Intrusion Prevention System for Windows Operating Systems. Springer-Verlag, Berlin Heidelberg.

Speice, C. 2003. Designing a Managed Antivirus Solution for a Large Corporate Environment. SANS Institute.

Tikkanen, A. 2010. Antivirus Engines Basics. F-Secure Corporation.

Received: Sept 19, 2013; Accepted: Oct 30, 2013