Florida International University
FIU Digital Commons

FIU Electronic Theses and Dissertations | University Graduate School

3-24-2008

A Behavior Based Approach to Virus Detection

Jose Andre Morales
Florida International University, jose@josemorales.org

DOI: 10.25148/etd.FI08081536
Follow this and additional works at: https://digitalcommons.fiu.edu/etd

Recommended Citation
Morales, Jose Andre, "A Behavior Based Approach to Virus Detection" (2008). *FIU Electronic Theses and Dissertations*. 41.
https://digitalcommons.fiu.edu/etd/41

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A BEHAVIOR BASED APPROACH TO VIRUS DETECTION

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Jose Andre Morales

2008

To: Interim Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Jose Andre Morales, and entitled A Behavior Based Approach to Virus Detection, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Xudong He

Geoffrey Smith

B.M. Golam Kibria

Peter J. Clarke, Co-Major Professor

Yi Deng, Co-Major Professor

Date of Defense: March 24, 2008

The dissertation of Jose Andre Morales is approved.

Interim Dean Amir Mirmiran
College of Engineering and Computing

Dean George Walker
University Graduate School

Florida International University, 2008

© Copyright 2008 by Jose Andre Morales

All rights reserved.

DEDICATION

I dedicate this dissertation to my family, loved ones and friends for giving me strength, patience, emotional balance and fortitude during the process of achieving this milestone in my life.

ACKNOWLEDGMENTS

First and foremost I wish to thank Dr. Yi Deng for taking the risk of believing in me and supporting me throughout my time as a graduate student. Special thanks to Dr. Peter J. Clarke for giving endless hours of time to my work providing invaluable feedback, comments, support and most importantly balance, strength and several cups of coffee. An oversees thanks to Dr. Eric Filiol for his countless reviews of my work and support of my research. A huge thanks and deep appreciation to Mario Consuegra for giving selflessly of his busy schedule to take the lead role in creating my implementation prototypes. A great thanks to the CIS faculty, especially Dr. Xudong He, Dr. Geoffrey Smith, Dr. Masoud Milani, and Dr. B.M. Golam Kibria from the statistics department for their tremendous help, feedback and advice. A big thanks to Tariq, Gonzalo, Irene, Rafael, Selim, Andrew, Yingbo, David and the rest of the ECS 2nd floor crew for helping me surpass the numerous barriers I encountered on the road to my PhD. My deepest appreciation to Martha, Haydee, Donaley, Olga and the rest of the CIS staff for putting up with all my questions and requests, never failing to help me when I needed it.

ABSTRACT OF THE DISSERTATION

A BEHAVIOR BASED APPROACH TO VIRUS DETECTION

by

Jose Andre Morales

Florida International University, 2008

Miami, Florida

Professor Peter J. Clarke, Co-Major Professor

Professor Yi Deng, Co-Major Professor

Fast spreading unknown viruses have caused major damage on computer systems upon their initial release. Current detection methods have lacked capabilities to detect unknown viruses quickly enough to avoid mass spreading and damage. This dissertation has presented a behavior based approach to detecting known and unknown viruses based on their attempt to replicate. Replication is the qualifying fundamental characteristic of a virus and is consistently present in all viruses making this approach applicable to viruses belonging to many classes and executing under several conditions. A form of replication called self-reference replication, (SR-replication), has been formalized as one main type of replication which specifically replicates by modifying or creating other files on a system to include the virus itself. This replication type was used to detect viruses attempting replication by referencing themselves which is a necessary step to successfully replicate files. The approach does not require a priori knowledge about known viruses. Detection was accomplished at runtime by monitoring currently executing processes attempting to replicate. Two implementation prototypes of the detection approach called SRRAT were created and tested on the Microsoft Windows operating systems focusing on the tracking of user mode Win32 API system calls and Kernel mode system services. The research results showed SR-replication capable of distinguishing between file infecting viruses and benign processes with little or no false positives and false negatives.

TABLE OF CONTENTS

CHAPTER PAGE

1 INTRODUCTION 1
1.1 Motivation 4
1.2 Research Problem 6
1.3 Proposed Solution 6
1.4 Novel Contributions 7
1.5 Scope and Limitations 9
1.6 Outline of the Dissertation 10

2 LITERATURE REVIEW 12
2.1 Background 12
2.1.1 Basic Definitions 12
2.1.2 Computer Viruses 13
2.1.3 Virus Detection 22
2.1.4 Windows API System Calls 26
2.1.5 File System Operations 27
2.2 Related Work 28

3 SELF-REFERENCE VIRUS REPLICATION 33
3.1 Formal Model 36
3.2 Detection Algorithms 39
3.3 Example 41
3.4 Limitations 42

4 SELF-REFERENCE DETECTION PROTOTYPE 45
4.1 User Mode Prototype 50
4.1.1 Implementation 53
4.1.2 Limitations 55
4.2 Kernel Mode Prototype 56
4.2.1 Implementation 58
4.2.2 Limitations 60

5 SELF-REFERENCE DETECTION EXPERIMENTS 62
5.1 Theory Validation 63
5.2 User Mode Prototype 63
5.3 Kernel Mode Prototype 64

6 TEST RESULTS: ANALYSIS AND EVALUATION 72
6.1 Theory Validation 72
6.2 User Mode Prototype 80
6.3 Kernel Mode Prototype 85
6.4 Evaluation of Proposed Solution 89

7 CONCLUSIONS AND FUTURE WORK 96
7.1 Conclusions 96
7.2 Future Work 97

BIBLIOGRAPHY 99

VITA 103

LIST OF FIGURES

FIGURE PAGE

3.1 56 Viruses with Replication Attempts 35

3.2 56 Benign processes with Replication attempts 36

3.3 Formal definition of SR property 37

3.4 Transitive relation of SR 38

3.5 Formal definition of $SR - replication$ 39

3.6 Sample Abstract Graph for vx1 40

3.7 Reorganized abstract graph for vx1 after removal of node M_2 41

3.8 $SR - replication$ of Cassidy Peer-to-Peer Worm 43

3.9 Win32 API calls with equivalent read/write operation 44

4.1 SRRAT Architecture 46

4.2 Mapping of Read Win32 API calls in user version SRRAT 51

4.3 Mapping of Write Win32 API calls used in user version SRRAT 52

4.4 List of Win32 API calls needed to implement user version of SRRAT . . . 53

4.5 Mapping of System Services used in Kernel Version SRRAT 58

5.1 Virus Classification with Total Samples amount 62

5.2 Theory Validation Test Benign Processes - 1 64

5.3 Theory Validation Test Benign Processes - 2 65

5.4 Theory Validation Test Viruses - 1 66

5.5 Theory Validation Test Viruses - 2 67

5.6 Test Viruses for User implementation of SRRAT 68

5.7 Test Viruses for Kernel implementation of SRRAT - 1 68

5.8 Test Viruses for Kernel implementation of SRRAT - 2 69

5.9 Test Viruses for Kernel implementation of SRRAT - 3 70

5.10 Test Viruses for Kernel implementation of SRRAT - 4 71

6.1 Theory Validation Test Results Benign Processes 74

6.2 Summary Results Theory Validation Virus Test 75

6.3 Theory Validation Test Results Viruses - 1 76

6.4 Theory Validation Test Results Viruses - 2 77

6.5 Theory Validation Test Results Viruses - 3 78

6.6 Theory Validation Test Results Viruses - 4 79

6.7 EW-Win32.Alanis SR-replication graph 81

6.8 Virus Test Results User implementation of SRRAT 82

6.9 Viruses not hooked by User implementation of SRRAT 83

6.10 Viruses not Exhibiting SR-replication in User Mode SRRAT Testing . . . 83

6.11 Summary Results Kernel Implementation SRRAT Virus Test 86

6.12 SRRAT Kernel Mode Log File Amus Virus 87

6.13 SRRAT Kernel Mode Log File Borzella Virus 87

6.14 Virus Test Results Kernel implementation of SRRAT - 1 91

6.15 Virus Test Results Kernel implementation of SRRAT - 2 92

6.16 Virus Test Results Kernel implementation of SRRAT - 3 93

6.17 Virus Test Results Kernel implementation of SRRAT - 4 94

6.18 Virus Test Results Kernel implementation of SRRAT - 5 95

LIST OF ACRONYMS

ISR	indirect self reference
SR	self reference
SRR	self reference replication
SRRAT	self reference replication analysis tool
VX	viruses

CHAPTER 1

INTRODUCTION

The dominance of networked computers and wide spread use of wireless devices has created a new fertile ground for novel creation and release of destructive, highly infectious, fast spreading computer viruses. Robust virus detection is essential to protect against these new unknown threats. Current virus detection technology though very effective in detecting previously discovered viruses, falls short of providing effective detection against just released unknown viruses. The main reason of this ineffectiveness against unknown viruses is the high dependence of a signature database as the centerpiece of detection for current virus detectors. Using a signature database only allows for detection of previously discovered viruses but fails in the detection of unknown viruses. A previously discovered virus has been isolated and examined by virus researches. A signature which is a unique string of bytes uniquely identifying a particular virus is extracted from the executable version of the virus and added to the database. If the virus evolves in some form that alters the portion of its executable form pertaining to the signature, detection of the evolved version of the virus may fail.

Behavior based virus detection is a promising approach capable of detecting unknown viruses. This form of virus detection does not use a signature database. Instead executing processes are monitored and their behavior is analyzed. If the execution behavior of a process exhibits virus behavior then the process can be flagged as being a possible virus. Virus behavior is predefined by the behavior based detection approach that is being implemented and is usually done by a knowledge expert in computer viruses. The main problem of behavior based virus detection is defining a virus behavior that assures the detection of both known and unknown viruses while not incorrectly detect benign processes as being a virus. The result of this diffi-

culty is several behavior based virus detection approaches define virus behavior that successfully detect viruses and not benign processes only under stringent execution conditions. Other behavior based detection approaches define virus behavior that successfully detect only a specific class of virus. All of these behavior based approaches are built upon some characteristic of a virus that is identifiable only under specific execution conditions or in a specific class of virus. To be more effective, these behavior based detectors should be based upon a virus characteristic that is consistently present in all viruses. This can facilitate successful detection of both known and unknown viruses belonging to several classes and under several execution conditions. One goal of this research is to identify a characteristic of computer viruses that is consistently present in all viruses and create an effective and robust behavior based virus detection approach that is based on that characteristic.

The seminal research formally defining a computer virus show the qualifying fundamental characteristic of a virus to be replication. When a malicious code is classified as a virus, it is done so only after assuring the malicious code has the ability to replicate. Replication is the main characteristic that differentiates a computer virus form other forms of malicious code. This makes replication the fundamental qualifying characteristic of a virus, thus every virus is guaranteed to have the ability to replicate. The implication of this is that replication is the one characteristic of a virus that is consistently present in all viruses regardless of what class the virus belongs to and regardless of under what execution conditions the virus will run. This makes replication an excellent virus characteristic that can be used to build an effective and robust virus detection approach. The virus detection approach presented in this research is based on virus replication.

A virus will execute a series of operations during its replication process. Having the ability to replicate does not guarantee a virus will successfully replicate every time it executes. A virus may not even attempt replication for several different reasons. When a virus does attempt replication, there is one property of this process that is almost always present. This property is implied in the following observation: *"When a virus replicates, what is it replicating?"*, the answer is *"itself"*. This observation implies an essential part of the replication process which is when a virus attempts to replicate itself it must refer to itself as the source of the replication. The act of a virus referring to itself in order to replicate is an essential property of virus replication. In this research we define this property to be the *self-reference property* (SR). When a virus attempts a replication where SR is present, we define that specific form of replication as *self-reference replication* (*SR-replication*).

In this research the term "itself" refers to the static image of the virus file saved on a storage device such as a hard drive. When a virus replicates it may do so in one of three general ways: it may copy itself into an already existing file, it may create a new file containing a copy of itself or it may copy itself into a memory location and continue execution from there. The first two types are part of a general class of viruses titled file infector viruses, the last type belongs to a virus class titled memory resident viruses. There are many known viruses that are strictly memory resident viruses, but they are outnumbered by the far larger class of file infector viruses. There are also several hybrid viruses that replicate as both file infector and memory resident. The vast majority of known viruses are either strictly or partially file infector viruses.

Another important aspect of virus replication is the desired destination, normally this is one of two places: the local computer where the virus is saved or executing from, or another computer across a network, where the network can be wired or wire-

less. A subtype of the virus called a worm is designed to replicate across networks, many classes exist for worms such as peer-to-peer worms, network worms and email worms. One of the goals of a worm is to replicate from one computer to another, but in many cases they will first replicate on the local computer to then attempt replication across a network to another computer. Most of the known worms replicate both on a local computer and across networks with a smaller number strictly replicating across networks. Viruses can spread via replication across computers and networks either slowly or very fast. Many new viruses have been termed fast spreading, these types of viruses such as Warhol worms and Fast worms can spread to over 90% of vulnerable computers in as little as 15 seconds. It is difficult to determine how many times a virus will attempt to replicate on a local computer. Static analysis of a virus could provide evidence hinting toward high or low frequency of replication attempts. Unfortunately in the case of fast spreading viruses, static analysis is not an option when immediate detection is of utmost importance.

This research is to develop a detection approach for file infector viruses attempting SR-replication on a local computer. An assumption is made that SR-replication is a behavior belonging to computer viruses and is unlikely to occur in benign processes. The goal of the detection approach is to stop the proliferation of known and unknown fast and slow spreading viruses on a local machine by terminating them upon identifying their first attempt to replicate using SR-replication where the virus can belong to one of many classes and can execute under several different conditions.

1.1 Motivation

Effective detection of unknown viruses upon first release is one of the biggest challenges facing computer virus researchers today. A study from 2006 showed that virus

companies required on average 6 hours to analyze and extract a signature from a newly discovered virus and add it to their signature database [56]. This was the same amount of time needed by these companies two years earlier in 2004 [8]. During the same two year period several fast spreading destructive viruses most notably the Witty worm [49] have emerged in the wild causing several millions of dollars in damages. In 2006, an FBI survey reported computer viruses as the number one cause of financial loss for American companies [28]. At over $67 billion dollars, viruses accounted for over 70% of all financial losses for the corporations surveyed. For the same year Kaspersky Labs reported a strong rise in the number of new viruses and more momentum in the second half of the year with email worms topping the list [4]. Kaspersky also forecasts that viruses will increasingly appear, helping to spread other forms of malware and use more sophisticated techniques to avoid detection. Virus writers are also being well funded by rogue governments and organzied crime, allowing the development of cutting edge detection avoidance and fast distribution techniques.

Despite this growing problem antivirus companies continue to use signature databases as their primary tool for virus detection. In 2006 Kaspersky labs averaged 10,000 new record updates to its signature database per month and 200 new malware samples per day [16]. Even after solutions have been released it is unknown how much time passes until all end user signature databases are updated. It is clear that antivirus companies will continue to improperly and slowly handle the ever growing virus problem using signature databases as the centerpiece of detection. The future outlook and forecast trends show viruses to be increasing and working closely with other forms of malware continuing to injure and infect computers world wide [3, 2]. In early 2008, the assistant secretary for cyber security and telecommunications at the Department of Homeland Security Gregory Garcia renewed the department's determination to

create a secure cyberspace citing the exponential growth of connected devices as a breeding ground of future attacks [32]. The history and future outlook of computer viruses on the global community along with the inadequate performance of current virus detection technologies serves has motivation for this research.

1.2 Research Problem

The research presented in [38, 5] demonstrated the inability of current signature based virus detectors to detect obfuscated versions of known viruses. This inability leaves computers susceptible to infection by unknown viruses. Currently, very few behavior based detection approaches are available and widely used as a main form of virus detection by companies and consumers. The central problem investigated by this research is the ability to find a characteristic of viruses that can be used to detect file infecting computer viruses when they first attempt replication in a local computer and can a differentiation be established between viruses and benign processes based on this characteristic. A second problem investigated by this research is the effectiveness of a behavior based virus detector using this characteristic as its main form of detection in stopping or minimizing the replication of both known and unknown file infecting viruses executing in a local computer with minimal false positives and false negatives.

1.3 Proposed Solution

In an attempt to solution the first research problem, SR-replication will be used as the main point of distinguishing between viruses and benign processes. By establishing this distinction the rate of false positives and false negatives between viruses and benign processes can be kept to a minimum. The second research problem will be addressed through the creation of an SR-replication based virus detector prototype that can be executed on a local computer to monitor processes as a real-time monitor.

When a process exhibits SR-replication behavior, that process is flagged as exhibiting possible virus behavior. The following assumptions are made for this solution:

1. SR-replication is a characteristic unique to virus and not observed in benign processes.
2. The prototype will only detect file infecting viruses which replicate using SR on a local computer.

To test the proposed solution, static analysis of known viruses will be conducted to analyze the presence of SR-replication. This analysis will also be conducted on benign processes to confirm our assumption by verifying the absence of of SR-replication in these non-viral processes. The prototype will be executed on a virtual machine and a real computer for several days to test for false positive production. The prototype will also be tested against a set of SR-replicating file infecting viruses to test for accurate detection and virus process termination plus the absence of false negative production. The effectiveness and resource usage of the prototype will be determined based on a several day test on a real computer with the goal being an establishment of non-overwhelming usage of system resources allowing the system to function in a normal manner while the detector is in use.

1.4 Novel Contributions

This research establishes three novel contributions in the area of computer virus detection, they are as follows:

1. Creation of a new virus detection approach using SR-replication.
2. Identification of known and unknown file infecting viruses with no a priori knowledge of known viruses using this approach that produces minimal false positives and false negatives.

3. A behavior based virus detector implementing the approach that is effectively used in a real time environment without significant system slowdown.

Up to this point there have been several behavior based approaches to virus detection. Many of these detection approaches use a specific characteristic of viruses for their detection methodology that is not consistently found in all viruses [39]. The result is detection limited to specific classes of viruses or under specific conditions. In this research I characterize a specific form of virus replication which is the fundamental virus characteristic present in all viruses. Establishing replication as a characteristic that is fundamental and present in all viruses makes it a perfect research area for new detection approaches. The benefit of a consistent characteristic is the the ability to rely upon it when creating detection algorithms. The reliability allows for the algorithms to possibly be very effective in detecting viruses belonging to several different classes and under many different execution conditions.

The ability of having one characteristic that identifies both known and unknown viruses with no a priori knowledge is a powerful step forward in the field of virus detection. The fact that this one characteristic also produces minimal false positives and false negatives makes it very appealing as a standalone implementation for detection in real time or as a component working with other establish detection methods resulting in in a more robust virus detection solution. A standalone implementation can consume minimal resources of the system and perform silently in the background this is ideal for home users. As one component of a bigger solution it is key for enterprises with a wealth of computer resources and are inclined to protect every vector of their system from a possible computer virus attack.

The principle benefit of a virus detector based on SR-replication is its ability to detect a virus replication upon its initial replication attempt in a computer system.

Many of the viruses released in recent history have been fast spreading viruses. These infect and injure a vast majority of susceptible computers in under 3 hours from initial release. A detector using SR-replication may be able to thwart some of this fast spreading by identifying the virus's replication behavior in a local computer and terminate it. Thus stopping that virus's proliferation into other computers across a system.

1.5 Scope and Limitations

There are two main forms of virus replication, the first is by infecting already existing files and creating new files containing a copy of the virus and the second is by creating a copy of the virus in memory. In its current form SR-replication will only detect viruses replicating by infecting existing files or creating new files which contain a copy of the virus. Viruses that replicate by creating a copy strictly in memory will not be detected. Besides replication, a virus also injures a computer in one or several different ways. SR-replication is designed to only detect the replication aspect of a virus.

If a virus injures and causes damage to a computer and never replicates it will not be detected by my approach in its current form. Also if a virus injures and causes damage to a computer before replicating it will not be detected until the replication commences, therefore any damaged incurred on a computer by a virus before replication can not be stopped or minimized by my approach. SR-replication is most effective in stopping or minimizing replication and damage from viruses that attempt replication when first executed or very early on in their execution.

In its current form SR-replication is designed to detect viruses replicating in a local computer and does not detect replication across a network from one computer to another. It will not detect viruses that strictly replicate across a network without ever attempting replication on a local computer at some point during its execution.

Viruses that replicate without SR will not be detected by my approach in its current form. It is possible for a virus to replicate itself by copying itself from some other location that already contains a copy of the virus. I define this form of replication as *indirect self-reference replication* (ISR-replication). This form of replication has been identified as part of the replication process of some viruses, for example the Anarch email worm and the Rega network worm [6, 46], but it is not commonly used by the vast majority of known viruses.

A prototype of SR-replication will have different levels of effectiveness depending on the specific layer of the operating system that is chosen as the target of the implementation. If a prototype is created at a user level it may have a smaller detection rate than a prototype implemented at a lower level, for example the kernel level. This results from the fact that viruses are designed to avoid detection as much as possible and very few may actually run at user level. The vast majority of viruses attempt to run at the kernel level or lower in an attempt to subvert higher level security measures that do not have privileged access to the lower levels of the operating system.

1.6 Outline of the Dissertation

The balance of this dissertation is as follows: Chapter 2 will provide background and related work, Chapter 3 presents the theoretical foundation for SR-replication, Chapter 4 is a description of the implementation prototypes used in this research,

Chapter 5 details the prototype experiments, Chapter 6 gives an analysis of testing results and Chapter 7 consists of conclusion and future work.

CHAPTER 2

LITERATURE REVIEW

2.1 Background

Society has been dealing with computer viruses since the early 1980's [44]. During this time over 70,000 viruses have been discovered in the wild and the number continues to grow. Understanding computer viruses is a never ending task that is necessarily part of being a computer virus researcher in order to keep up with detection of the latest virus writer techniques. The evolution of viruses in this time period has been extraordinary with the development of almost foolproof detection avoidance techniques, fast spreading distributions, innovative injure and infection strategies and novel approaches of attacking vulnerable computers. In this chapter I will start with some basic definitions then present background information on computer viruses and detection focusing on theoretical foundations along with different classifications of virus types, different forms of file infection strategies and virus evolution techniques. An overview of file system operations and Windows API system calls, which are used in the implementation prototypes will also be given. Related work to this research will also be presented along with the betterments of this research in comparison to these noted works.

2.1.1 Basic Definitions

1. Malware: a portmanteau of the words Malicious and Software, is defined as software that is created to injure and/or infect a computer system without the express consent of the computer's owner. Malware has several subcategories including: viruses, worms, trojan horses, logic bombs, mass-mailers, germs, exploits, downloaders, dialers, droppers, flooders, rootkits, adware and spyware amongst others.

2. Virus: a program that infects other programs by modifying them to include a possibly evolved version of itself.

3. Worm: a specific form of a virus designed to replicate across networks.

4. Trojan Horse: a program attempting to appear as a legitimate benign program that once executed performs benign actions while covertly performing malicious actions.

5. False Positive: A condition that arises when a benign object is classified as a malware.

6. False Negative: A condition that arises when a malware is not identified and classified as benign.

2.1.2 Computer Viruses

One of the early works on computer security threats is presented in [31]. Anderson presents a definition for a threat as being a potential possibility of a deliberate and unauthorized attempt to access or manipulate information or render a system unreliable or unusable. He defines vulnerability as a hardware or software flaw that can lead to accidental disclosure of information. An attack is a planned execution of a threat and a penetration is a successful attack. Anderson describes how surveillance of users activity, via log records, on a system can identify computer threats. He defines external and internal penetrations giving different levels or rings or security to protect data from being penetrated. Three types of users are defined for detecting internal penetration: 1. the masquerader, 2. the legitimate user, 3. the clandestine user.

The masquerader is seen as the easiest to detect by identifying abnormalities in time of computer use, frequency of use, amount of data access and which data was

attempted to be accessed. Since the legitimate user is authorized to use the system, abnormal usage patterns become harder to detect. Anderson claims that abnormalities in totals of time of computer use, data access, attempts to access unauthorized data would need to be analyzed to detect the misfeasance of a legitimate user. The clandestine user is argued as the most difficult to detect assuming that administrative rights have been acquired. This allows for activities that may not be recorded by log records. Also the ability to erase/overwrite/deactivate log records may be available within the privileges. Anderson presents a surveillance system that includes file monitoring, user monitoring and device monitoring all based on analysis of log records and analyzing for abnormalities in time, frequency, access and other parameters. The definitions of threat, vulnerability, attack and penetration given by Anderson is embodied in the currently accepted definition for malware [62]. His methods of detecting attacks and penetrations by analyzing log records is seen in the publications of others including those dealing with intrusion detection [47, 12, 45]

A computer virus is defined as a program that can infect other programs by modifying them to include a possibly evolved version of itself [22]. The key characteristic of a virus is its ability to replicate. A virus must replicate either by infecting other files or producing possibly evolved copies of itself. A mailicious program that does not replicate can not be classified as a virus. Computer viruses have become very sophisticated in detection avoidance, fast spreading and causing damage. A highly populated taxonomy of viruses exists with each classification having its own challenges for successful detection and removal [14, 44]. Today viruses are regarded as a real global threat and viewed as a weapon usable by those bent on creating large scale interruption of everyday life [11, 48].

The concept of self-replication was considered by John Von Neumann and Arthur Burks [29, 61]. Their works based on cellular automata essentially proved the replication property can be realized. Though virus replication was not explicitly addressed in their work it laid foundation for future seminal papers on viruses. The automata produced were very complex, others produced less complex self reproducing automata and can be found in [18, 10, 27, 40]. In 1938, Kleen published the Recursion Theorem [50]. This is considered the very first formalism of a self reproducing program. The theory proves the existence of viral programs through the use of recursive functions. The theorem also describes what is later to be called polymorphic viruses.

Thompson presents in [57] a method of showing how a trusted program can in fact abuse that trust by creating self reproducing programs that are deemed to be benign along with creating other trusted programs that are in reality Trojan Horses. Thompson creates a C compiler that is modified to purposely create malicious versions of itself. This is accomplished by knowing when it is compiling a new version of itself at which point the malicious code is implanted. This new version of the C compiler will forever create malicious versions of itself and other C compilers every time they are compiled using the malicious compiler. Thompson also shows how the compiler can implant a Trojan horse in other programs by identifying them at compilation time. The point is illustrated with the Unix login program that is implanted with malicious code allowing another person to log into any Unix system using that login program. The moral of this Turing Award winning paper is no program that was not fully written by you can never be fully trusted. Interestingly enough the form in which the C compiler functions with the malicous code makes it a form of a self replicating program.

In 1986 Fred Cohen defended his PhD dissertation at the University of Southern California entitled Computer Viruses [20]. This dissertation is considered one of the seminal papers researching computer viruses. It was the first time the term virus was used to represent self replicating programs. Using Turing machines, Cohen describes a virus as being a sequence of symbols which is able, when interpreted in a suitable environment, to modify other sequences of symbols in that environment by including a possibly evolved copy of itself. He also provides the first formal definition of a computer virus using Turing machines.

A virus is presented as being a viral set, where membership is acquired if and only if a sequence of symbols on an input tape causes a Turing machine to copy a sequence of symbols to some new location further down on the tape. This captures the fundamental meaning of virus replication. Cohen also gives a formal definition for viral evolution, showing that a virus can change its appearance during replication. This was the fundamental definition for the class of polymorphic and metamorphic viruses [44]. Many properties of viral sets are formally shown. He proved that it is undecidable to determine if a virus is an evolved version of some other virus.

In [21], Cohen conducts various experiments running computer viruses on computer systems. The results showed that systems protected by the Bell-LaPadula [13] system were insufficient to prevent viral attack and determines integrity control [34] to be essential to securing a system. Cohen states in [21] that absolute protection from viruses is attainable by absolute isolation of a computer but notes that this is impractical. He further states that to be secure from viruses a system must protect against outgoing information flow and to be secure from information leakage a system must protect against incoming information flow. To allow sharing there must be information flow between computers in a system. Thus Cohen's main conclu-

sion in [21] is sharing in a general purpose multilevel security system is in such direct contrast to virus security that they are not reconcilable and coexistence is impossible.

A second seminal paper in the area of computer viruses was presented by Adleman [36]. Adleman built from the work of Cohen's PhD dissertation, Adleman was his adviser at USC [14]. Adleman considered the more general class of malware instead of focusing on viruses. He gave classifications for malware and formal definitions and proofs based on recursive functions. He states for every program there is an infected form of the program. A virus can be viewed as a mapping from non-infected to infected programs. An infected program on every input will do one of three things: injure by damaging the computer, infect by replicating itself into other programs or imitate where it neither infects nor injures. Imitate is considered a special case of injure when no files are found suitable to infect.

The two basic types of malware defined are pathogenic and contagious. Where pathogenic will injure and not infect or imitate. Contagious will infect or imitate but not injure. The following four basic types are defined: benignant is not pathogenic or contagious, Trojan horse is pathogenic and is not contagious, carrier is not pathogenic and is contagious, and virulent is pathogenic and is contagious. This classification considers the behavior of an infected program compared to its non infected form. A second classification is given, here only the behavior of an infected program alone is considered. Four types are defined as follows: benign is not pathogenic and not contagious, Epeian is pathogenic and not contagious, disseminating is not pathogenic and is contagious, and malicious is pathogenic and contagious.

Three final observations are made with members of the second class and those of the first class as follows: if an object is benign then it is benignant, if an object is

Epeian then it is benignant and a Trojan horse, and finally if an object disseminating then it is benignant and a carrier. The second classification included all current forms of malware with Epeian includes non-replicating malware such as Trojan horses and logic bombs [15, 44] and disseminating and malicious encapsulate replicating malware such as viruses and worms. The classifications are built to map every classification back to a logical connection of the three basic behaviors of an infected program: injure, infect and imitate. Today classification of malware is still performed using these same basic characteristics.

Viruses have several subcategories of which the most prevalent are email worms, peer to peer worms, network worms and Win32 viruses [44]. Email worms attempt to spread primarily by covertly sending out emails attaching the virus or a copy of the virus to the email. The addresses are usually harvested from the victim computer and the victim's email address is used as the sender, this is done to add a layer of legitimacy to the virus and give confidence to the recipient in opening the attachment thus causing the virus to enter a new computer and continue on its path.

Peer to peer worms are designed to spread through networks via peer to peer file sharing software such as Limewire, Emule, Morphius and others. Often these worms will replicate in known shared directories of a computer by creating new copies of itself several times. Each new copy will have a filename reflective of a currently popular song, video game, celebrity or movie in the hopes of enticing other users to download and execute the file on their computer.

Network worms replicate primarily by exploiting a vulnerability in a computer system allowing them to spread from one computer to another across several systems. Win32 viruses represent those viruses that employ the Microsoft Windows

32bit Application Program Interface (Win32 API) of the Windows operating system as part of their execution. Viruses using the Win32 API at some point during their execution are the most prevalent amongst all viruses [63].

There is a wide array of methods in which a virus can infect a file. Each of these is a specific form of either modifying an already existing file are creating a new file which is simply a copy of the virus itself. What follows is a list of the better known infection strategies along with a brief description of each.

- Overwriting Viruses: a primitive technique that simply overwrites existing files with a copy of the virus. Files that fall victim to this infection cannot be disinfected. They must be erased and restored from an existing uninfected backup copy, an example of this virus is the Loveletter mass mailing email worm. This virus spread across network via attachments to emails, once executed on a computer the virus would overwrite with itself all files ending in one of a predefined set of file extensions.

- Appending Viruses: infect existing files by inserting a copy of the virus at the end of the file. To successfully execute, the virus will modify the header portion of the infected file by inserting a jump command to the memory location of the virus within the file. The result of this is the virus executing first each time the infected file is accessed. These files can be disinfected. The Vienna virus implements this infection technique.

- Prepending Viruses: A very successful technique, this infection works by the virus inserting itself into the beginning of a already existing file. The result is instant execution of the virus each time the infected file is accessed. The file can be disinfected, a good example of this technique is the virus Polimer.512.A

which inserts itself with a size of 512 bytes long to the beginning of executable (.EXE) files.

- Parasitic Viruses: Similar to the prepending virus, this type of infection inserts the virus to the beginning of an already existing file and appends the original beginning of the file to the end of it. In some cases the entire original file will be save as a newly created file in some other location of the computer. A file infected in this way is difficult to disinfect, normally the correct order of the file cannot be restored. Viruses using this technique are Virdem, Jerusalem, Qpa and Klez.

- Cavity Viruses: This technique does not increase the size of the file it is infecting. It works by inserting the virus into useless areas of an existing file, for example a file may contain an area full of zero's, the virus will target this area to insert itself. At the beginning of the file the virus inserts a jump command to the memory location of the beginning of the virus. At the end of the virus is a second jump command back to the original start point of the file. Some examples of this technique are Lehigh and W2K.Installer.

- Entry Point Obfuscation (EPO) Viruses: This infection technique inserts the virus within some area of an already existing file. A jump command to the virus location is not placed at the beginning of the file, instead it is placed at some other location within the body of the file. This leads to the virus being executed at random. It is also a form of avoiding certain detection techniques that analyze the beginning of files for possible modifications like the insertion of a jump command. Some viruses using this technique are Olivia and Nexiv-Der.

Among the many different classifications in which viruses can be placed, one important category is code evolution techniques. Virus evolution occurs during the

replication process of a virus. The purpose of evolution is to avoid detection by creating new versions of the virus that appear differently is some form when compared to the parent virus that created it. There are three basic categories of virus evolution: Non-evolving viruses, Polymorphic viruses and Metamorphic viruses.

Non-Evolving viruses do not modify their appearance in any way when they enter the replication process. Every time the virus replicates, the newly created virus appears exactly the same as the parent virus. The impact of this is facilitation of identifying the virus on a computer system. A signature based virus detector only needs one signature to detect all copies of the virus in a given system.

Polymorphic viruses mutate in some form while keeping the original virus code intact. This is usually achieved through encryption, where a virus creates an encryption shell around itself hiding the virus from detectors. Many of these viruses carry with them a decryption engine which is usually not encrypted and can be used for detection. More advanced viruses also encrypt the decryption engine to further avoid detection. In every case of polymorphic viruses there is always a small portion that is not encrypted and is the target of detection. Some known polymorphic viruses are 1260, HPS and Marburg.

Metamorphic viruses mutate their appearance by actually changing the virus code itself. Encryption is not used to hide the virus code, instead the virus code itself is modified to produce a different appearing virus that performs the same as the parent. This type of virus is normally a single piece of code that carries data in the form of variables within it. When executed the virus uses this data in a decision process to created evolved versions that have some its code replaced or augmented with the data. The result is the same virus in a new body. There is great difficulty in

detecting this type of virus since with every evolution the signature used for detection can potentially be completely replaced with new code.

2.1.3 Virus Detection

Virus detection can be defined as the ability to identify the presence of a virus in an object [22, 44]. The single most important aspect of Cohen's work in [22] is his analysis of virus detection. He proves the problem of viral detection to be undecidable via a reduction from the Halting problem [22, 21]. The implication of this result is no viral detection algorithm is capable of detecting all known and unknown viruses. This leads to the production of false positives which are non viral objects being identified as viral and false negatives being those viral objects not being identified as such.

Many virus detection algorithms have been presented [43], each with its advantages and disadvantages. Virus detection can be classified as one of two forms: signature based and behavior based [44, 14]. Signature based detectors work by searching through an object for a specific sequence of bytes that uniquely identify a specific version of a virus. This form of detection is also known as string scanning, it is the simplest form of scanning and is quite effective. This type of detection relies on a virus signature database [15, 44].

Each time a new virus is discovered in the wild, the binary form of the virus is analyzed by a virus researcher. One of the goals of the analysis is to find a sequence of bytes in the binary that can be used to specifically and uniquely identify the specific virus being analyzed. The sequence of bytes is copied from the binary and added to the virus database. The sequence of bytes used to identify a virus is called the virus signature. Virus detectors using these databases must be updated frequently to get the latest database with the most recently added virus signatures. Currently signa-

ture based virus detection is the predominant detection method used in computers around the world.

Behavior based detectors identify an object as being viral or not by scrutinizing the execution behavior of a program [23, 47, 44, 15]. These detectors do not use signatures to identify a specific virus. Instead they use measures of normal or abnormal behavior to detect a running processes's behavior as viral. Abnormal behavior is flagged as viral and is either terminated or suspended from executing further. Behavior based detection is able to detect unknown viruses [1, 17, 9]. Since the behavior of a viral process can be similar to that of benign processes, behavior based detection can cause false positives and false negatives. The overhead of dealing with production of false positives keeps behavior based detection currently out of wide spread computer use [1, 17].

In [19], Ellis presented a detection method for worms in a network using behavioral signatures. Ellis defines a behavioral signature as having aspects of any particular worms behavior common across the manifestations of a specific worm and with nodes that spread in temporal order. Three characteristic patterns of worm behavior in a network were identified as: passing similar data between two machines, Identification of the tree-like structure created by the intercommunication patterns that emerge from infected nodes and changing a server into a client. These behaviors were developed from the definition of a worm. Ellis also introduces network application architecture (NAA) as a method for distributing network applications. NAA impacts the sensitivity of this behavioral approach. This is done by placing constraints on traffic patterns, which are violated by the patterns of worm traffic. These constraint violations are shown to be straightforward to detect. The abstract communications network model (ACN)is a network theoretical model of computer networks and their

data flow presented in this paper. It is a realistic network including representations of hosts, routers, sensors, hardware, routing, data flow, spans, user workstations and servers. The behavioral signatures, NAA's and worm propagation are all implemented within the ACN framework. A worm propagation model, consisting of a set of spanning trees that extends the ACN to include worm spread across a network is given. A descendant relation between nodes in the spanning trees captures communication patterns that appear as a result of worm spread, this gives the foundation for detection. The paper identifies architectural designs that improve worm detection sensitivity. The detection approach was shown capable of detecting classes of worms without a priori knowledge of any specific worm.

Schneider presents in [23] a characterization of security policies that are enforceable using execution monitoring which is an implementation of behavior based virus detection. A security policy specifies execution that is deemed undesirable. Execution monitoring can only enforce security policies that are a safety property. A policy that is a safety property will not allow any bad thing to happen. This means that when a bad thing is about to happen the enforcement mechanism terminates the execution. A security violation is to occur at the start of some execution. This approach does not know what may happen at some future point in the execution, it only knows what has happened. This could lead to false positive production by terminating an execution that violated a security policy and is a part of some bigger execution that does not violate any security policy. This condition greatly limits the type of monitoring performed. A security automata is characterized as the enforcement implementation of execution monitoring. When an execution starts an automata is initialized incorporating the security policy being enforced. For every step of execution the automata attempts a transition to an acceptable state using an input symbol coming from the executing object. If this succeeds the execution continues one more step. This back

and forth process continues until execution terminates or the automata enters and unacceptable state. This indicates a security violation and the execution should be terminated.

Bergeron presents [26] a method of detecting malicious code in commercial off the shelf components (COTS) using binary executable static slicing and statistical analysis. Static slicing is used to extract security critical code fragments. The fragments are then verified against behavioral specifications to statically decide if they include malicious behavior. The approach taken consists of three main steps: first, disassemble the binary code; second, create a high level abstract view of the disassembled code and use program slicing to remove parts of the code that are security critical; third, detect malicious behavior in the slices based on program checking. The disassembly is performed with commercial disassemblers. High level representation is achieved using transformations representing idioms. An idiom is defined as a sequence of instructions holding logical meaning not derivable from the individual instructions. Slicing the high level representation of the program is meant to retain only relevant instructions, particularly API calls that influence the value of registers. This is achieved by using a slicing criterion. The criterion specifies a node of a control flow graph of the disassembled code and a subset of program variables. The result is a set of instructions used in the computation of the variables subset, this is called a slice. By examining the variables modified in a given slice, the authors assume the slice to be malicious or not. A given example deals with a representation of passing information of a file named "security.txt". This leads to assumption of malicious behavior due to transmission of security critical information.

2.1.4 Windows API System Calls

With the release of Windows 95 by Microsoft in 1995, a new set of system calls was introduced. This set was call the Windows 32 bit Application Program Interface (Win32 API) [63, 44]. The set consists of 32 bit system functions usable by any Win32 application. The purpose of the API was to provide a set of optimized system level operations allowing applications to run faster. The set is currently supported by all Windows platforms. All the API functions are stored in the following dynamic link libraries: Kernel32.dll, User32.dll, Gui32.dll and Advapi.dll. When an process is labeled a Win32 process it indicates that process uses the Win32 API. When a Win32 process is first executed it is analyzed by the operating system and the memory address of each Win32 API system functions that it may call is exported from a DLL and placed in an import address table (IAT). Each Win32 process has its own IAT and when the process makes an API system call, it looks up the function's address in the IAT and passes to that address any necessary parameters and the function proceeds with execution. When an system call is made it is usually from a process running in User mode, the called function is filtered through the operating system to its equivalent function in the Kernel of the operating system. Once in the kernel a service is usually requested to carry out the operation and the result filters back up the the user application that originally made the call [25].

The Windows Kernel is an area of privileged access in charge of running the majority of system services. The kernel provides a set of its own Native Kernel mode API functions a subset of which is called Zwxx routines [41]. These routines can be called directly by any Kernel mode process. When one of these routines is called it is filtered to a system request to fulfill the task.

The complete list of Kernel mode functions is stored by memory location addresses in the System Service Dispatch Table (SSDT), this table is accessed each time a Zwxx routine is called, the parameters are then passed to the memory location and the function continues with its execution [25, 58].

2.1.5 File System Operations

A file can be considered as an abstract data type that has attributes and operations. The attributes of a file include: name, identifier, type, location, size, protection, and time, date and user identification [51]. The basic operations of a file include: creating, writing, reading, repositioning, deleting and truncating [24, 51]. A virus is defined as a program that can infect other programs by modifying them to include possibly evolved version of itself [22]. From the point of view of the system a virus is a file and therefore possesses the attributes and operations of files. I can deduce that if the virus copies itself is must therefore invoke the read and write file operations when it is infecting other programs. Therefore the virus must have the appropriate access privileges in order to perform the copy [35]. In my approach it does not matter if the copy was successful or not since I am just interested in the virus making an attempt to replicate.

In this research I use the name, identifier and location file attributes to reference the static image of the file on a storage device. The name (identifier - a unique tag) of a file F is represented as $F.name$. The location of F is usually an argument of the write and/or read operations that are used during file replication. Writing F involves making a system call specifying both the name of F and the location where F will be written. To read F a system call is invoked that states the name of F and where in memory F or a part of F will be placed. In the event that F cannot be written or

read in one execution of the operation then a pointer keeps track of the next block to be written or read.

2.2 Related Work

Skormin et al. present an approach to detect replication in self contained propagating malware [52]. Their detection is done by monitoring at run-time the execution of normal code under regular conditions. They monitor the behavior of each process and analyze the system calls, input and output arguments and the execution results. The Gene of Self Replication models the replication of a process using building blocks. Each block is a portion of the self replication process including opening, closing, reading, writing and searching for files and directories. The approach detected several viruses across many classes with little or no false positives. My detection method focuses only on read and write operations that have SR. This is a simplification of the Skormin et. al. approach which consider additional operations such as search, open, create as essential parts of a replication process. My simplified approach reduces the overhead time and analysis needed to detect virus replication resulting in faster detection.

Analysis of system call arguments to detect malicious attacks is found in [12]. Several models are presented to characterize system call arguments. These characterizations are used to detect anomalous behavior. The research states two assumptions: (1) malicious attacks appear in system call arguments. (2) system call arguments used in malicious attacks substantially differ from arguments used during normal application execution. The models detect anomalies in the arguments such as unreasonably long string length, unusual characters and illegitimate values. The analysis of the arguments are used to create a score that determines if the system call is part of an attack. The models were trained with sequences of system calls giving no regard to

the sequence but focusing only on the arguments. The testing results showed the models to be effective in detecting malicious attacks with low false positives. My research also analyzes system call arguments without considering the sequence in which the system calls are made. The difference in my approach is I only consider read and write system calls used during replication of a virus. My analysis of system calls is much faster since it only involves a simple check for SR-replication. This faster approach results in less overhead quicker detection than what Mutz et al. propose.

A rule based approach for analyzing system call arguments and their invoking process appear in [37]. The research proposes a threat level classification of system calls based on their ability to penetrate a system with full system control and denial of service attacks. This classification is used to grant invocation of security critical system calls. The system call is invoked only if the process invoking the call and the arguments comply with an access control database. The system call arguments are analyzed to detect malicious intent such as rewriting a critical directory or process. This work takes file system calls into special consideration as they can lead to penetration of privileged areas of a system. My work also considers file system calls as they are primarily used in the replication process of a virus. My approach differs in which system call arguments are analyze. I only analyze arguments of read and write system calls leading to less analysis and faster detection.

The host-based intrusion detection system BlueBox [53] defines system call introspection. The introspection consists of rules used to analyze system calls when they are invoked to conclude if they can be part of an intrusion or not. System call arguments are scrutinized to prevent time-of-check-to-time-of-use attacks and proves effective. The arguments are recorded by BlueBox for a system call at time of check. At the time of use the arguments present in the system call are compared to the

recorded arguments. Any difference in the arguments concludes the system call to be a possible attack. My approach also analyzes system call arguments, but at the point of execution. I inspect the arguments of system calls that are a read or write system call. My approach has the advantage of analyzing the arguments of a smaller number of system calls leading to faster analysis and detection.

In [47], anomalous intrusion detection was performed via system call monitoring. A database was trained to recognize the normal behavior of benign processes in a system. The benign processes were executed multiple times and their sequence of system calls was recorded in the database. Anomaly detection occurred by monitoring process executions. The system calls made by the process were compared to the database, if the process made system calls not matching the database, the process was marked as anomalous. An assumption was made stating anomalous behavior is assumed as an intrusion. Their are some key drawbacks to this approach. First, determining exhaustive training of the database for a benign process. It is the decision of a human to state when a benign program has been executed enough times to capture all possible sequences of system calls even with short sequences. What guarantee can be given toward a short sequence of system calls not being left out? If this occurs the detector can create a false positive by identifying normal behavior as anomalous. Second, frequent retraining is needed each time a new program is added to the system. Each time a new program is installed on the system the database must be trained to recognize its normal behavior. This adds overhead to the maintenance of the system, if one program is left out of training the protection of the system is at risk. Third, the paper does not address training a database for normal behavior of a program that has an intrusion built into it. In this case the intrusion will be recognized as normal behavior and a false negative will be produced. My research takes the same approach of monitoring system calls to detect intrusions. The difference of

my approach is the analysis is done on sequence of system calls done by viruses. More specifically, the subset representing the replication of the virus. The analysis is done without a priori knowledge resulting in no need for a detection database. This is an improvement on the approach in [47] where training must be done and a database updated for each new program.

In [45], finite state automata (FSA) is used to train normal behavior of programs in a system. The training results is stored in a database later used for detection of intrusions seen as anomalous behavior. The training works by recording the sequence of system calls of a benign process after multiple executions. During monitoring the FSA attempts to transition from one state to the next based on the system calls made by the executing process. If the FSA cannot perform a transition or encounters an unknown location of a system call the process is marked as anomalous. If a transition occurs but enters a state not in the automaton the FSA enters a sink state. This is a temporary state that allows execution of new code. The FSA will transition out of the sink state when able to enter a new state found in the automaton. A leaky bucket algorithm is used to aggregate anomalies over time. Weights are given to anomalies based on the seriousness of the specific anomaly. The weights are designed to flag a process as an intrusion when anomalies have occurred a certain amount of times. The more serious anomalies weights cause in intrusion detection in as little as one occurrence. There is one critical drawback to this approach. Allowing new code to execute while the FSA is in a sink state can allow for an intrusion to occur and not be flagged as anomalous. The intrusion can be dynamically injected into the process at runtime and execute freely, thus creating a false negative. Also an encrypted intrusion such as a polymorphic virus can bypass detection. The sink state will allow viral code that is unencrypted at runtime to execute. Upon completing execution the viral code returns control to the host at which point the FSA would exit the sink state

without detecting anything as anomalous. The research I am conducting follows the idea of weights for detecting viruses. I similarly use metrics when analyzing system calls of an executing process to detect viral behavior. Since I detect viral behavior based on replication, a virus that is unencrypted or dynamically injected and executed during runtime should be detected by my implementation. This is true since being classified as a virus implies replication will occur at some point in the execution.

CHAPTER 3

SELF-REFERENCE VIRUS REPLICATION

Replication is the fundamental qualifying characteristic of all viruses [22, 14, 44]. For a specific malware to be classified as a virus it must have the ability to replicate. This guarantees the replication characteristic is consistently present in all viruses. Replication is therefore an excellent basis for detection algorithms to successfully detect viruses under several conditions and that belong to many different classes [30]. When a virus replicates, it will execute a series of operations that will cause the virus to be written to some other area of the target system. The virus can infect one or more currently existing files and infect the system by copying itself to newly created target files. Both of these infection types require a series of read and write operations to succeed.

Self-reference is an essential property of the read and write operations executed by a virus during replication. A virus must refer to itself in order to replicate itself to some other area of the target system. The term “itself” refers to the static image of the virus file saved on a storage device such as a hard drive. The name of the virus file is the same as the name of the executing virus process. This name is passed between read and write operations as the source or ”from” argument of the replication. I named this property the self-reference property (SR) and replication that occurs using SR I identify as SR-replication. SR is the focus of this research and SR-replication is the centerpiece of my behavior based virus detection approach. The basic definition of SR-replication is a running process transferring and storing data, belonging to its source object, from one object to another object where the first transfer is always from the source object. The object is any temporary or permanent storage available in a computer and accessible by a running process. This includes

both file and memory storage. The source object is from where the process was created. For examples a running process's source object may be the static image file that is executed to create the process. The process can transfer data from any object including its source object to any other object as long as the data being transferred belongs to the source object of the process performing the transfer. I present a detection approach for SR-replication that is based on SR which focuses on the transitive relation between a running virus's static image file and a target file. The transitive relation is based on the transfer of data from the virus's static image file to the target file. By detecting SR-replication I may be able to detect both known and unknown viruses belonging to different virus classes and that execute under several conditions. I further assume SR-replication to be unique to viruses and that it is unlikely for SR-replication to occur in benign processes. I do recognize that not all viruses will replicate using SR-replication and these viruses may not be detected by my approach.

Static analysis of viruses and benign processes was conducted to establish preliminary support on my assumptions of SR-replication. A test set of 56 viruses was built by downloading live samples from various Internet malware repositories [60, 42]. A second test set of benign processes was built using 56 executable processes from the Microsoft Windows System32 folder. All the viruses were randomly chosen and belong to the classes of Win32 viruses, network worms, email worms and peer-to-peer worms. The virtual machine software VMware Workstation with Windows XP SP2 was used to execute the the test sets. The programs Api Spy 32 and Process Monitor [7, 55] were used to create log files documenting the system calls made by each process in one complete execution. Each log file was examined for SR-replication. This was determined through identification of SR by examining the arguments of read and write system calls for a reference in the "from" argument that was the name of the currently executing process or a temporary memory location where the currently

Email Worms	**Replication Attempts**	**Peer to Peer Worms**	**Replication Attempts**
Baconex	**1**	Agobot.a	**1**
Bagle.a	**1**	Banuris.b	**217**
Bagle.j	**1**	Bereb.a	**474**
Bagle.k	**1**	Bereb.b	**481**
Bagle.m	**1**	Blaxe	**6**
Bagle.n	**1**	Cassidy	**19**
Bagle.o	**1**	Cocker	**61**
Dumaru.r	**3**	Compux.a	**36**
Eyeveg.m	**1**	Delf.a	**1**
Klez.a	**3**	Gagse	**257**
Klez.e	**1**	Irkaz	**2**
Klez.i	**1**	Kanyak.a	**1**
klez.j	**2**	Kifie.c	**2**
Mimail.j	**1**	Mantas	**233**
Network Worms	**Replication Attempts**	**Win32 Viruses**	**Replication Attempts**
Afire.b	**3**	Apathy.5378	**1**
Afire.d	**1**	Arch.a	**1**
Bobic.k	**1**	Barcos.a	**4**
Bozori.b	**1**	BCB.a	**60**
Bozori.e	**1**	Bee	**2**
Bozori.j	**1**	Canbis.a	**14**
Cycle.a	**1**	Civut.a	**1**
Dabber.c	**1**	Cornad	**1**
Domwoot	**1**	Jlok	**2**
Doomjuice.b	**1**	Parite.a	**1**
Doomran	**1**	Parite.b	**1**
Incef.b	**27**	Tenga.a	**1**
Kidala.a	**1**	Watcher.a	**1**
Lebreat.a	**1**	Zori.a	**1**

Figure 3.1: 56 Viruses with Replication Attempts

executing process had copied itself earlier in the execution. The results of the testing are in Figures 3 and 3.2.

The total number of SR-replication for each process listed in Figures 3 and 3.2 is the count of distinct filenames that each process attempted to infect in one execution. I did not verify if each attempt was a success or a failure. The attempt to perform SR-replication is enough for me to label the process as a possible virus regardless if it is successful or not. The test results all 56 viruses attempted SR-replication at

Benign Processes	**Replication Attempts**	**Benign Processes**	**Replication Attempts**
accevt	**0**	ckcnv	**0**
accwiz	**0**	cleanmgr	**0**
actmovie	**0**	clipbrd	**0**
ahui	**0**	cmd	**0**
append	**0**	cmdl32	**0**
blastcln	**0**	common32	**0**
bootcfg	**0**	control	**0**
bootok	**0**	convert	**0**
cacls	**0**	cscript	**0**
charmap	**0**	csrss	**0**
chkdsk	**0**	ctfmon	**0**
chkntfs	**0**	debug	**0**
cipher	**0**	defrag	**0**
cisvc	**0**	diskpart	**0**
Benign Processes	**Replication Attempts**	**Benign Processes**	**Replication Attempts**
diskperf	**0**	ipconfig	**0**
dllhost	**0**	ipv6	**0**
dmremote	**0**	lodctr	**0**
doskey	**0**	lpq	**0**
eventcreate	**0**	lsass	**0**
exe2bin	**0**	makecab	**0**
extrac32	**0**	mem	**0**
fastopen	**0**	netsetup	**0**
finger	**0**	notepad	**0**
fsutil	**0**	ntbackup	**0**
getmac	**0**	openfiles	**0**
help	**0**	ping	**0**
hostname	**0**	qprocess	**0**
iexpress	**0**	setup	**0**

Figure 3.2: 56 Benign processes with Replication attempts

least one time to as many as over 400 times in a single complete execution. None of the benign processes attempted SR-replication. These results provided support of my assumptions and lead me to create a formal model for SR-replication.

3.1 Formal Model

An operation o is invoked with arguments $(a_1 \ldots a_n)$ by a currently executing process P where $P.name$ is the name of P. The static file image F saved on a stor-

age device is from where P was created. The name and path of F is held in $F.name$ and $P.name \mapsto F.name$, thus $P.name$ refers both to P and F. The label T is a temporary memory location containing a copy of F. When an operation $o \in O = \{read(s,d), write(s,d)\}$ where the *source* argument $s = a_i$ and *destination* argument $d = a_j$ with $1 \leq i,j \leq n$ and $i \neq j$ is invoked by P where $s \in S = \{P.name, T\}$ then o is said to have the *self-reference property* (SR). The argument $d \in D = \{M, I.name\}$ where M is temporary memory location and $I.name$ is the name of the destination static image file I saved on a storage device with $I.name \neq P.name$. The formal definition for SR is given in Figure 3.3.

$SR(o) = true$ iff $o \in O$ and $o.s \in S$ with

- $O = \{read(s,d), write(s,d)\}$
- $s \in S = \{P.name, T\}$
- $d \in D = \{M, I.name\}$
- $P.name$ = name of currently executing process that is invoking o
- T = temporary memory location containing a copy of the static file image F
- M = temporary memory location
- $I.name$ = name of the destination file
- $o.s$ = the s argument of o
- $o.d$ = the d argument of o

Figure 3.3: Formal definition of SR property

I restrict the set O to only read and write operations. I assume a process only needs to execute a sequence of these two operations to attempt replication. The sets S and D are restricted to static file images and temporary memory locations because I am only detecting replication of one file to one or more files where one or more temporary memory locations are used to complete the process. The basis case for

$SR(o) = true$ is with $o.s = P.name$. In this case P refers to F in an attempt to read or write itself to $o.d$. In the case where $o.s = T$, $SR(o) = true$ when $o(T, d)$ was invoked by P at time t, $o(s, M)$ was invoked by P at time t', $t' < t$ and $T = M = F$. In this case P must have previously invoked at least one o with $o.d = M$, placing F into M which results in M converting to T. By uniquely enumerating all o executed by P with $1 \leq m \leq n$, I can define $SR(o_m)$ in terms of $F\mathtt{R}o_m.s$ as shown in Figure 3.4. Testing for $SR(o_m)$ is equivalent to establishing a *transitive relation* $\mathtt{R}$ between F and $o_m.s$. When $F\mathtt{R}o_m.s = true \rightarrow F = o_m.s$ through invocation of $o_1 \ldots o_m$ by P.

$$\forall o_m(s, d) \text{ executed by } P \text{ with } 1 \leq m \leq n,\ SR(o_m(s, d)) = true \text{ iff } F\mathtt{R}o_m.s = true$$

Figure 3.4: Transitive relation of SR

P invokes a sequence of o_m operations with $1 \leq m \leq n$. If $o_1.s \in S$, $o_m.d = I.name$, $o = write(s, d)$, $I.name \neq P.name$ and $SR(F, I) = true$ then P is said to have performed *self-reference replication* (SR-replication). The formal definition of SR-replication in Figure 3.5 focuses on detecting processes that read and write their static file image to other newly created or already existing static file images. This can be accomplished in one write operation or in several read and write operations, also many memory locations can be used intermediately from F to I. $SR(F, I)$ is established by testing for SR on every o that leads from $P.name$ to $I.name$, thus $SR - replication(P) = true$ iff a transitive relation $F\mathtt{R}I$ = true. I assume that static file images can only be read from and written to. The definition does not detect a process that overwrites or modifies its own static file image.

$SR - replication(P) = true$ iff

- $\exists o_1 \dots o_m$ with $1 \leq m \leq n$, where
  - $o = write(s, d)$ and
  - $o_m.s \in S$
  - $o_m.d = I.name$
  - $I.name \neq P.name$
  - $SR(F, I) = true$

Figure 3.5: Formal definition of $SR - replication$

3.2 Detection Algorithms

When P starts execution, the operations o can be traced using a *directed graph* G consisting of $edge = o_m$ and $node = \{P.name, T, M, I.name\}$. A graph is created for each P in a system and is linked to a specific P by the value of the first node of G which must always be $P.name$. Upon P invoking its first operation o where $o_m.s = P.name$ a new G is created and its $root\ node = P.name$. When a new edge is added it must be of the form $o_m.s \rightarrow o_m.d$ with $s \in S$ and $d \in D$ and the value $o_m.s$ must already be present as a previous $o_m.d$ node in G with exception of cases where $o_m.s = P.name$ which is the root node of G. A sample graph is given in Figure 3.6 for a process named $vx1$.

In Figure 3.6 each o is enumerated in order of execution by P. The first two operations $read(M_1, M_3)$, $write(M_3, sys.bat)$ are not included in the G since neither has $o_m.s = P.name$ which is vx1. The root node of the G must always be the first o of P where $o.s = P.name$. I see this in $read_1$ where $read_1.s = vx1$. Notice the operation $read_{1,6}$, the notation shows the operation with the same arguments occurred twice, at the first and sixth invocation. Every operation in G is true for SR and correctly placed in the form $o_m.s \rightarrow o_m d$. A test for $SR - replication(vx1)$ was done when the operation $write_5(M_2, services.exe)$ was added to G. The path vx1 *rightarrow*

- $P.name = vx1$
- operations = $read(M_1, M_3), write(M_3, sys.bat), read_1(vx1, M_2), read_2(vx1, M_4), read_3(M_2, M_5), read_4(M_5, M_6), write_5(M_2, services.exe), read_6(vx1, M_2)$

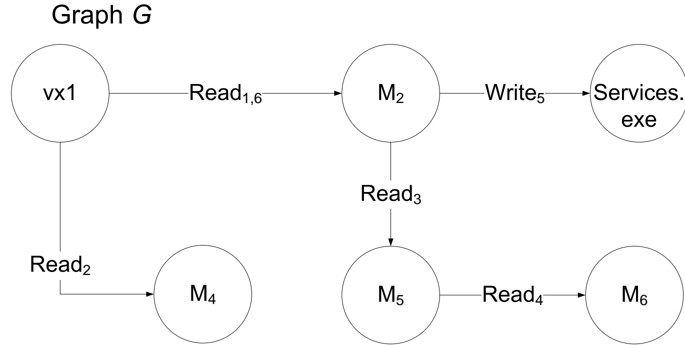

Figure 3.6: Sample Abstract Graph for vx1

services.exe shows the transitive relation $F\mathtt{R}I$. This path also satisfies my definition of $SR-replication$ in Figure 3.5 and therefore $SR-replication(vx1) = true$. When a graph G of a process P contains a path from $P.name \rightarrow I.name$ then $F\mathtt{R}I = true$ which results in $SR-replication(P) = true$. Construction of G only continues until $SR-replication(P) = true$ then P can be flagged as exhibiting virus replication. If P finish execution and $SR-replication(P) = false$ then P is assumed benign.

If P invokes an operation $o_m(s, d)$ where $SR(O) = false$ and $o_m.d$ is already a node of G, then $o_m.d$ must be removed in one of two ways: If $o_m.d$ is a leaf node, it is simply removed and G remains the same. If $o_m.d$ is an internal node in G then $o_m.d$ is removed and G is reorganized by eliminating all incoming edges to $o_m.d$ and repositioning all outgoing edges from $o_m.d$ to each child node to come from each parent node of $o_m.d$ to the child node. Figure 3.7 shows graph G from Figure 3.6 after removal of node M_2. The incoming edge $Read_{1,6}$ from the parent node $vx1$ was eliminated and the outgoing edges $Read_4$ and $Write_5$ were each reposition to come

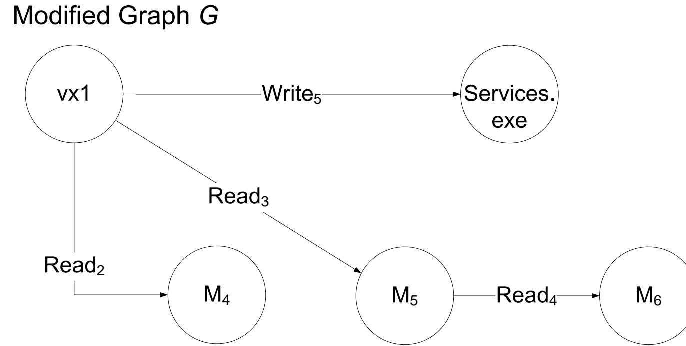

Figure 3.7: Reorganized abstract graph for vx1 after removal of node M_2

from the parent node $vx1$ to the child nodes M_6 and $services.exe$.

3.3 Example

In this section I will use portions of the log file of a virus used in the static analysis to give an example of SR and $SR - replication$ using a graph for testing. The log file was created using API SPY 32 [7] which logs all the Win32 API calls invoked by a process [41, 63]. The example in Figure 3.8 is of the Cassidy worm, a packed Peer-to-Peer worm [54, 44] that from the static analysis testing results in Table 3 attempted replication 19 times. In the partial log file, the Cassidy worm attempted to copy itself six times using the API call `CopyFileA` which is the same as the API call `CopyFile` but is used when dealing with the ANSI character set [63]. From Table 3.9, `CopyFileA` is mapped to $write$(`lpExistingFileName`,`lpNewFileName`). As an example, the fourth `CopyFileA` operation is mapped to:

$write$("C:\DOCUME 1\JAM-VX 1\Desktop\CASSIDY.EXE",
"C:\WINDOWS\Shared Folder\kazaa hack.exe").

All the other operations are mapped in similar fashion. In the graph:
$rootnode$ = CASSIDY.EXE and $SR(o_m) = true$ for each o_m in the graph.

Consider
$write_4$(C:\DOCUME 1\JAM-VX 1\Desktop\CASSIDY.EXE,
C:\WINDOWS\Shared Folder\kazaa hack.exe).

We can see:
P =CASSIDY.EXE,
$P.name = write_4.s$ = C:\DOCUME 1\JAM-VX 1\Desktop\CASSIDY.EXE and
$I.name = write_4.d$ = C:\WINDOWS\Shared Folder\kazaa hack.exe.

Applying these values to the definition of SR in Figure 3.3, results in $SR(write_4) = true$ and this result holds for all the other $write_m$ operations as well. When operation $write_1$ was invoked, the graph was updated and a test for $SR - replication$ was conducted since a $write$ operation occurred with $write.d = I.name$ = diablo 2 pindlebot.exe. It is clear to see that $SR - replication(CASSIDY.EXE) = true$ according to the definition in Figure 3.5,. Had this been a real time detection, the process would have been flagged as exhibiting virus replication behavior. To allow readability, only the filenames were placed in the graph of Figure 3.8 when it should be the complete path and filename.

3.4 Limitations

My approach is based on general read and write operations. I assume any specific operation that performs a read, write or copy by specifying in the arguments the source and destination can be equivalently written using the general read and write operations used in this research. Figure 3.9 shows some Win32 API calls [63] and

Partial Log File for Cassidy Worm

```
00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\diablo 2 pindlebot.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\diablo 2 pindlebot.exe",
                              DWORD:00000104)

00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\diablo 2 maphack.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\diablo 2 maphack.exe",
                              DWORD:00000104)

00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\playstation2 emulator.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\playstation2 emulator.exe",
                              DWORD:00000104)

00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\kazaa hack.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\kazaa hack.exe",
                              DWORD:00000104)

00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\cable modem utility.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\cable modem utility.exe",
                              DWORD:00000104)
```

Cassidy SR-replication graph

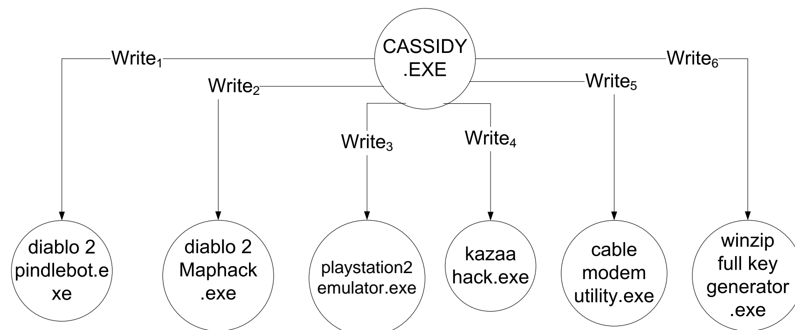

Figure 3.8: $SR-replication$ of Cassidy Peer-to-Peer Worm

their conversion to an equivalent general read or write operation. Note that I am only interested in the source and destination arguments of the operation.

My approach focuses on detecting $SR-replication$ on a local machine, it currently does not detect $SR-replication$ from one local machine to another across a network. I am aware of the ability of some viruses to replicate without using $SR-replication$. This can be accomplished either by replicating from a source that is not P or invoking commands in some other process that results in replicating P. These types of

Win32 API	Read/Write operation

`BOOL WINAPI CopyFile(`
`__in LPCTSTR lpExistingFileName,`
`__in LPCTSTR lpNewFileName,`
`__in BOOL bFailIfExists);`
	write(`lpExistingFileName`, `lpNewFileName`)

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

`BOOL WINAPI WriteFileEx(`
`__in HANDLE hFile,`
`__in LPCVOID lpBuffer,`
`__in DWORD nNumberOfBytesToWrite,`
`__in LPOVERLAPPED lpOverlapped,`
`__in LPOVERLAPPED_COMPLETION`
`_ROUTINE lpCompletionRoutine);`
	write(`lpBuffer`, `hFile`)

Figure 3.9: Win32 API calls with equivalent read/write operation

replication I refer to as indirect self-reference replication, $(ISR - replication)$, and is currently not detectable by my current approach.

CHAPTER 4

SELF-REFERENCE DETECTION PROTOTYPE

To test my SR-replication theory, a runtime monitor implementation prototype named SRRAT (SR-Replication Analysis Tool) was created in two versions. One version runs in user mode and the other in Kernel mode, both prototypes were built to run on the Windows XP platform. The user mode version tracks API function calls and the Kernel version tracks system services used by all currently running processes using a technique known as hooking [25, 58]. Each prototype followed the design architecture in Figure 4.1. The architecture consists of two main components: API Call Processor and the SR-Replication Detector. The API call processor is composed of: HookAPI, MapAPI-RW and an API Repository. The SR-Replication Detector consists of: SR test, SR-Replication test, Update-Graph and a graph storage.

The overall idea of the prototype is to follow the execution of processes on a system. As the process executes it will inevitably interact with the operating system and this interaction is recorded and analyzed by SRRAT. The method of interaction between all processes including viruses and the operating system is through the invocation of API function calls and Kernel system services [25, 58, 63, 41]. SRRAT tracks only a subset, principally those that implement file system operations: open, close, read, write, copy and and a few other operations. When one of these is invoked by a process, SRRAT hooks it and analyzes its parameters to determine the presence of SR and if SR-replication has occurred. A hook is a method by which a user can redefine a standard API call and have the operating system redirect invocations of the standard API call to the user defined API call.

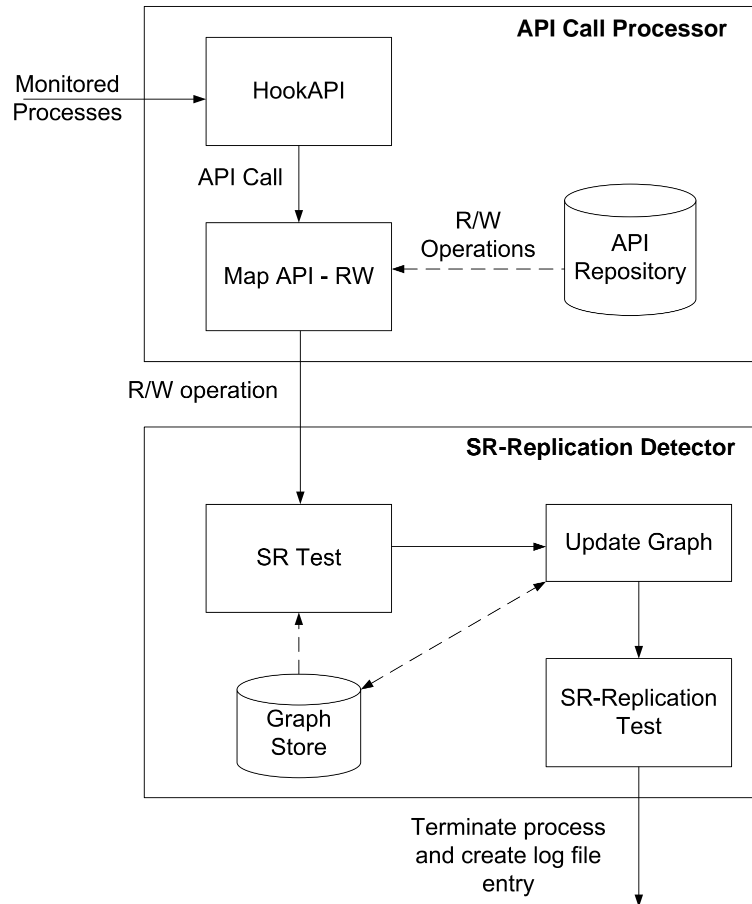

Figure 4.1: SRRAT Architecture

It is very powerful in the sense that a user can modify the execution behavior of processes without having direct access to the source code of that process. The following is a description of the purpose and responsibility of each component of SRRAT.

The API Call Processor's (ACP) main purpose is to detect the invocation of an monitored API call and pass its parameters to the SR-replication detector. The ACP is in idle mode waiting for the operating system to send a notification that a monitored API has been invoked by some process. At this point the APC takes control of the invocation and checks to see if the API is a read or write operation according to a predefined mapping. If the API is a read or write operation the APC passes it

along with its parameters to the SR-replication detector for further processing. During this time the process that originally invoked the API is in a wait state pending the completion of the API. This serves to stall the execution of possible viruses while they are being analyzed for SR-replication. The APC consists of two subcomponents called HookAPI and MapAPI-RW plus an API repository which are explained next.

The HookAPI subcomponent of the ACP is responsible for the actual interception of the API calls being monitored by SRRAT. The interception is done using an API hooking mechanism that notifies the ACP of the invocation of a specific API call which HookAPI has hooked. Once the process calls an API that has been hooked, the operating system redirects the call from the standard API to the user defined API where the redirection is part of HookAPI. When HookAPI completes its job the standard API call and its parameters have been redirected to the user defined API call and thus commences the second component of ACP which is MapAPI-RW.

The second subcomponent of the ACP is called MapAPI-RW and it serves the singular purpose of deciding if the API call that has been passed to it is a general read or write operation. If the API is determined to be a read or write operation then the API is labeled as such and it is passed along with the source and destination parameters to the SR-replication detector, which is the second component of SR-RAT. The determination of an API being a read or write operation is accomplished by searching for the API name in the API repository and checking if its mapping is to a read or write operation. If it is matched to a read or write operation, the API parameters specified in the repository as the source and destination parameters are parsed from the API call's parameter structure and passed along with the API and its read/write label to the SR-replication detector.

The ACP has an API repository which is a list of all the hooked API functions. The list has the API function name and the parameter names of the source and destination parameter according the the specific API function's documentation [63, 41] along with its mapping as a read or write operation. The API repository does not have the name of API functions that are not read or write operations even though they may be hooked by SRRAT for various implementation reasons. Only those API functions that represent a read or write operation require a mapping to a general read or write operation with the appropriate parameters and therefore are the only ones that need to be stored in the repository.

The second main component of SRRAT is the SR-replication detector (SRD). This component will execute as a result of the ACP passing along to SRD an API function that has been determined to be a read or write operation. The API function is received with the function name, a read or write label and the source and destination parameters. SRD has several responsibilities, the first one is to check for SR, if SR has occurred then a graph has to be created for the process that invoked this API function. If a graph already exists it is updated. The second responsibility is to check for SR-replication, this is done when a graph is updated with a write operation where the destination is the name of a file. The third responsibility is to return a detection confirmed message back to SRRAT so the process can be terminated and flagged as exhibiting possible virus behavior. The read and write operations of a process are stored in a graph using nodes and edges. SR-replication is determined by traversing the graph to establish transitivity between the process name at the root node and a file name located in some leaf node. SRD is composed of three subsections: SR-test, SR-replication test and update graph plus a graph storage which are explained below.

The SR-test subcomponent of the SRD is responsible for testing if a process has attempted to reference itself in the source parameter of an API function that it has invoked. I am most interested in this case when it occurs in read or write operations. The test is performed by comparing the process name with the source parameter of the API function. If the process name is a substring of the source parameter then the process has tested positive for SR. The other form of testing for SR is to search the existing graph of the process for a node that matches the source parameter of the API function. If a match is made on the graph then process has tested positive for SR. When SR occurs the API function with all its parameter information is passed along to update graph for insertion in an existing graph or creation of a new graph.

The second subcomponent of the SRD is called SR-replication test (SRT) and its principle responsibility is to check if SR-replication has been attempted by a specific process. This test occurs every time a process's graph has been updated with a write operation where the destination is a file. The graph is traversed backwards from the just inserted node, which contains the destination parameter of the API function which is a file name and path, back to the root node of the graph. If a path exists between these two points then the transitivity property holds true between the process and another file and therefore SR-replication has been attempted and SRT returns true to SRRAT.

The third subcomponent of SRD is Update-Graph which is in charge of adding new nodes to the graph as they are passed in from the SR-test. When an API function with its source and destination parameter are passed in, one of several actions can be taken. If there is no existing graph for the process that invoked this API function and the source parameter is the file name and path of the process, then a new graph is created with the source parameter as the root. If a graph already exists

for the process, the graph is traversed to find a node that matches the API functions source parameter. When a match is made if the node has no outgoing edges or none of its outgoing edges point to a node that matches the destination parameter then a new edge is created from the existing node to a new node with the file name and path stored in the destination parameter. If an edge already exists with the same source and destination parameters but a different API function name on the edge, the new edge is created. If an edge already exists with the same source and destination parameters and the same API function name on the edge then only its enumeration is modified to show the order of execution for multiple attempts of the same API function by the same process with the same source and destination parameters. Once the update to the graph is done, a notification is sent to SRT if and only if the just inserted edge contains an API function that is a write operation. The last operation done by this subcomponent before exiting is to save the graph in the Graph Store.

The SRD has a Graph Store which is a temporary memory storage of all the graphs currently being used by SRRAT to track processes. Each graph is accessed by the root node which holds the name of the process currently running on the system. When a process with a graph in the store finishes execution or is terminated by SRRAT, its graph is destroyed to release memory and reduce resource usage on the system.

4.1 User Mode Prototype

The first version of SRRAT was implemented as a user mode process running in Windows XP. In this version, SRRAT traced the Win32 API function calls invoked by all currently running user mode processes. The prototype was a terminate and stay resident runtime monitor, meaning it would be quietly running in the background monitoring the execution behavior of all user mode processes currently running on

Win32 API	Read operation

`void CopyMemory(`
` PVOID Destination,`
` const VOID* Source,`
` SIZE_T Length);`
	read(`Source`, `Destination`)

`BOOL WINAPI ReadFile(`
` HANDLE hFile,`
` LPVOID lpBuffer,`
` DWORD nNumberOfBytesToRead,`
` LPDWORD lpNumberOfBytesRead,`
` LPOVERLAPPED lpOverlapped);`
	read(`hFile`, `lpBuffer`)

`BOOL WINAPI ReadFileEx(`
` HANDLE hFile,`
` LPVOID lpBuffer,`
` DWORD nNumberOfBytesToRead,`
` LPOVERLAPPED lpOverlapped,`
` LPOVERLAPPED_COMPLETION_ROUTINE`
`lpCompletionRoutine);`
	read(`hFile`, `lpBuffer`)

Figure 4.2: Mapping of Read Win32 API calls in user version SRRAT

the system while conveniently placed as an icon in the windows task bar for simple start and stop functionality. The API functions that were traced for read and write operations are listed in Figures 4.2 and tableofmappedwriteapis along with their read-/write mapping and the source and destination parameters. All the API functions that were monitored by SRRAT are located in the kernel32.dll dynamic link library. SRRAT implemented API hooking on the functions that were being monitored. To successfully perform hooking SRRAT was implemented as a dynamic link library. Aside from the API functions monitored in Figures 4.2 and 4.3 for their read and write operations necessary to establish SR and SR-replication, there were other API functions, listed in Figure 4.4 that had to be hooked and monitored to correctly implement this version of SRRAT. The following is a description of the implementation of the components of SRRAT in user mode.

Win32 API	Write operation
BOOL WINAPI CopyFile(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName, BOOL bFailIfExists);	*write*(lpExistingFileName, lpNewFileName)
BOOL WINAPI CopyFileA(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName, BOOL bFailIfExists);	*map*(lpExistingFileName, lpNewFileName)
static BOOL WINAPI CopyFileW(LPCWSTR lpExistingFileName, LPCWSTR lpNewFileName, BOOL bFailIfExists);	*map*(lpExistingFileName, lpNewFileName)
BOOL WINAPI ReplaceFile(LPCTSTR lpReplacedFileName, LPCTSTR lpReplacementFileName, LPCTSTR lpBackupFileName, DWORD dwReplaceFlags, LPVOID lpExclude, LPVOID lpReserved);	*write*(lpReplacementFileName, lpReplacedFileName)
BOOL WINAPI WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped);	*write*(lpBuffer, hFile)
BOOL WINAPI WriteFileEx(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite, LPOVERLAPPED lpOverlapped, LPOVERLAPPED_COMPLETION lpCompletionRoutine);	*write*(lpBuffer, hFile)

Figure 4.3: Mapping of Write Win32 API calls used in user version SRRAT

Win32 API	Read/Write operation
`void CreateFileW( LPCWSTR lpFileName);`	update list with new file/handle: `lpFileName` and `HANDLE`
`HANDLE WINAPI CreateFileA( LPCSTR lpFileName);`	update list with new file/handle: `lpFileName` and `HANDLE`
`HFILE WINAPI OpenFile( LPCSTR lpFileName);`	update list with new file/handle: `lpFileName` and `HFILE`
`BOOL WINAPI CloseHandle( HANDLE hObject);`	*read*(`hFile`, `lpBuffer`)
`BOOL WINAPI DeleteFile( LPCTSTR lpFileName);`	remove from list existing file/handle: `lpFileName`

Figure 4.4: List of Win32 API calls needed to implement user version of SRRAT

4.1.1 Implementation

The HookAPI was implemented using hooking techniques for Win32 user mode API function calls. This is accomplished with the invocation by HookAPI of an API function called `SetWindowsHookEX` [63, 41]. Invoking this function allowed SRRAT to hook API functions by rewriting the IAT of all currently running processes. SRRAT reads from the API repository all the API function names that needed to be monitored which are listed in Figures 4.2 and 4.4. These names are loaded into memory and HookAPI invokes `SetWindowsHookEX`. For each API functions that needs to be hooked, SRRAT has a new version of the API which implemented my SR-replication detection code. Each running process has in its IAT table the memory address of all the Win32 API functions that it may invoke during its execution. When the hooks are placed for SRRAT, Windows overwrites these memory locations with memory locations of my redefined API functions. This change of memory addresses allowed Windows to redirect invocations of the monitored API functions from the standard API function to my version of the function. Hooking the API functions in HookAPI was the critical step needed for SRRAT to function.

The MapAPI-RW subcomponent was automated as a result of API hooking techniques. My redefined API functions are executed only when the API is invoked by some process. As a result when the body of my API function executed I already predetermined the function to be a read or write. In the body of the function I inserted code to read the correct parameters that represented the source and destination and continued to the SRD component of SRRAT.

One key piece of information needed for the SRD to work was acquiring the name of the process invoking the API function currently being processed through SRRAT. In Windows, process names are represented in two forms, the first is as a string containing the full process name and path, the second is with a process id (PID). When a process starts execution, Windows assigns it an integer value which is the process's PID, this value is used for various tasks throughout the life cycle of the process, especially when it interacts with the Windows operating system. When a process invokes a hooked API, Windows provided SRRAT a field of the hooked API structure containing the full process name and path as a string. This was used to implement the SRD.

A second key piece of information used by SRD is names of files that are read or written by a process. in similar fashion to processes, files in Windows are represented in two forms, the first is a string with the file name and path, the second is with a file handle. The file handle is an integer value assigned to a file when it is first opened by a process and destroyed when the file is closed. Some API functions such as `ReadFile` and `WriteFile`, use handles to represent the file that is being accessed but other API functions such as `OpenFile` use a string to represent the filename. Thus one file can have two ways of representation and this required the SRD to keep a list of file names and their associated handle. Each time the API functions `OpenFile` and `CreateFile`

was invoked, SRD would make a new entry in the list of the newly opened or created file and the associated file handle. This entry would later be removed from the list when `CloseFile` was invoked. Each time `ReadFile`, `WriteFile` functions were invoked SRD would look up the file handle and return the associated file name and path.

Once all the needed information became accessible to the SRD, performing SR tests was straightforward, every time a read operation occurred the filename and path in the source parameter would be compared to the process name. If the process name was a substring of the source parameter then SR was indeed present. Each time SR was found the graph for the process would be updated in Update-Graph. The graph store was a set of graphs each one with a different process name as the root node which identified the graph uniquely. A test for SR-replication occurred each time the `WriteFile` or `CopyFile` API function was hooked. The graph for the process was retrieved from the storage and traversed backwards from the newly added destination node from the `WriteFile` or `CopyFile` API to the root node checking for transitivity. When SR-replication was established for a specific process, that process was terminated by SRD using the `TerminateProcess` API function [63, 41].

4.1.2 Limitations

Several viruses do not interact with the operating system at the user mode level. Instead, they deal directly with the kernel and its function calls thus completely avoiding SRRAT. These viruses cannot be detected by the user mode version of SRRAT. Also this version ran in user mode and can be infected by the very same virus it is trying to detect if the virus injured before infecting, thus rendering SRRAT useless. Some viruses purposely encrypt the parameters when calling an API, SRRAT would process these parameters correctly but can produce a false negative since the encrypted parameters do not match the actual files being manipulated by the virus.

Other viruses call the API functions directly by loading the dynamic link library and acquiring the memory addresses of the API functions needed for the virus to run. Later in execution the virus passes the parameters to the memory location, this approach completely avoids the hooks placed by SRRAT and are never detected.

4.2 Kernel Mode Prototype

The second version of SRRAT was created to run in Windows Kernel mode. This version traced the Zwxx system services provided by the ntdll.dll dynamic link library which are exported from the Kernel process named ntoskrnl.exe. These are all Kernel system services and can be called directly by a Kernel process or indirectly by a user mode process. When a user mode process invokes a user mode API such as `OpenFile`the API function sends the request from user mode to a system service in Kernel mode, in this case `ZwOpenFile`. Tracing Kernel mode system services has three main advantages over tracing user mode API function calls:

- This version allows high probability of identifying SR-replication that may have been missed by the user mode version. This results from the higher level of difficulty any process, including viruses, faces in trying to execute and avoid using Kernel mode system services. It is very difficult to avoid interacting with the Kernel in some form, thus the probability of detecting SR-replication.
- SRRAT itself has a higher level of protection from virus infection by running in the Kernel. The Kernel is considered to be privileged access and not every process including other Kernel processes can have direct access to this privileged space. This give SRRAT a higher rate of survivability from virus attack and therefore increases the chances of running longer and detecting SR-replication.
- The form in which Zwxx system services are structured requires two parameters to be included that represent the file name with path and the file handle. This

requirements eliminates the need to hook and open or close system services and also eliminates the need of SRRAT to keep a list of file names and handles. This reduction in processing allows SRRAT to run faster and with less consumption of processing time.

To run a process in Kernel mode it must be executed by loading it as a system service. Two separate programs were created for this purpose, one to load and the other to unload the service from the operating system. Normally, services do not have user mode start and stop functions, these were added for convenience. Also SRRAT in this version was purposely created as a rootkit [58, 25] to include some techniques allowing SRRAT to hide from the system and therefore avoid being attacked or infected by a virus. The two techniques used for this purpose was: 1. the system service was hidden from the operating system, the name of the service would not show as currently running by any application in Windows and 2. the configuration file used by the service was redirected to a different part of the operating system thus hiding it as well. These are only basic hiding techniques but in fact are very useful. They allow SRRAT to run on the system as an invisible process which puts SRRAT on the same level playing field as some advance viruses which is necessary for any virus detector to run effectively and successfully.

Hooking system services has a different implementation than hooking Win32 API function calls but the underlying theory is the same. In the Kernel all the system services including the Zwxx services are exported by their memory location to a table called the System Service Dispatch Table (SSDT). When a request for a service comes in from user mode, the specific request is located in the SSDT and the operating system carries out the service. Similarly to hooking Win32 API function calls, this version of SRRAT had a list of redefined system services, when SRRAT was loaded it would overwrite in the SSDT the memory location of the standard system services

Kernel System Service	Read/Write operation

`NTSTATUS ZwCreateSection()`
`OUT PHANDLE SectionHandle,`
`IN ACCESS_MASK DesiredAccess,`
`IN HANDLE FileHandle OPTIONAL);`
	$read$(`FileHandle`, `SectionHandle`)

`NTSTATUS ZwReadFile(`
`IN HANDLE FileHandle,`
`OUT PVOID Buffer,`
`IN ULONG Length,`
`IN PULONG Key OPTIONAL);`
	$read$(`FileHandle`, `Buffer`)

`NTSTATUS ZwMapViewOfSection()`
`IN HANDLE SectionHandle,`
`IN HANDLE ProcessHandle,`
`IN OUT PVOID BaseAddress,`
`IN ULONG AllocationType,`
`IN ULONG Win32Protect);`
	$write$(`SectionHandle`, `BaseAddress`)

`NTSTATUS ZwWriteFile(`
`IN HANDLE FileHandle,`
`IN PVOID Buffer,`
`IN ULONG Length,`
`IN PULONG Key OPTIONAL);`
	$write$(`Buffer`, `FileHandle`)

Figure 4.5: Mapping of System Services used in Kernel Version SRRAT

with my redefined versions and thus the requests would be redirected to SRRAT and its redefined versions, this is how SRRAT hooked the needed Zwxx system services, which are listed in Figure 4.5. The following is a description of the implementation of the components of SRRAT in Kernel mode.

4.2.1 Implementation

Overall implementation in this version of SRRAT was much easier than the user mode version for both the ACP and the SRD. HookAPI was implemented by modifying the memory address of standard system services with my redefined versions in the SSDT. Since there is only one SSDT for the entire operating system, the actual hooking only had to occur by SRRAT once as opposed to the user mode version where hooking occurred once for each process. The number of system services hooked were less than those in the user mode version, they primarily were only the Zwxx system

services that represented a read or write operation. In Kernel mode system services decompose user mode file operations to basic read and write operations. Most notably the Win32 API function call `CopyFile` is translated to a call to `ZwReadFile` and `ZwWriteFile` in Kernel mode. This decomposition to simplified services greatly reduced the implementation of HookAPI.

As was the case in the user mode version of SRRAT, the MapAPI-RW subcomponent was automated as a result of system services hooking techniques. My redefined system services are executed only when the Kernel receives a request for a specific system service. As a result when the body of my system service executed I had already predetermined the service to be a read or write. The services's body had code inserted to read the necessary parameters that represented the source and destination parameters of the service and SRRAT continued to the SRD component of SRRAT. In this version of the ACP, the API repository had a much smaller list of system service functions needed to be hooked. This was a key advantage allowing for slightly less memory usage when SRRAT was operating.

The SRD was implemented in principally the same fashion as in the user mode of SRRAT. Both the test for SR and the SRT subcomponent has the same basic code as their user mode version. One difference was the removal of the list of file names and handles which was needed in the user mode version for the SRD to work properly. In Kernel mode the system services already provide all the file information that had to be found by SRD in user mode. This slight reduction in code creates a faster implementation which is key when dealing with aggressive fast spreading viruses. The graph store was kept in Kernel memory and the individual graphs were created, destroyed and accessed in the same manner as the user mode version. When

a process was found to have exhibited SR-replication it was terminated using the Kernel system service `ZwTerminateProcess`.

4.2.2 Limitations

The main limitation with this approach was in its implementation. Kernel mode programming is a very complex form of programming with little available documentation to aide the programmer. Many of the types, structures, functions found in the Kernel are not documented and using them with some confidence is only based on feedback from other programmers that have walked the path before. Implementing most of the code took some researching before being successful. Only with experience can a programmer become skilled in working with the Kernel. Most of the implementation was built using already established code heavily modified and questions posted on various forums provided some answers and partial solutions, the rest was done through trial and error. Many of the undocumented code used in this version had to be modified or rewritten to please the build environment into successfully compiling and building the executable version of SRRAT.

Two main limitations encountered during implementation was: 1. obtaining the name of the process requesting the system service, which is critical to test for SR, graph creation and identification and 2. Acquiring enough memory for SRRAT to effectively run. It was very difficult at first to obtain working code that would produce a string representing the name of the process. After 4 days of trial and error the name was finally obtained. The Windows Kernel seemingly runs within a limited memory space called pools and all kernel processes use this memory pool for their specific purposes. Allocating and using Kernel memory is a difficult science to understand and implement. Several setbacks were suffered by SRRAT using too much memory causing Windows to display a blue error screen, also known as the Blue Screen of Death

(BSOD), which led to the system crashing and requiring a restart. The main memory problems came with building the graphs in the SRD which works by implementing a linked list, each pointer was created with a chunk of kernel memory, it seemed this process repeated several times caused the system to produce the BSOD. The memory problem was not solved and this resulted in the implementation creating log files displaying all of the needed information to detect SR-replication in a given process.

CHAPTER 5

SELF-REFERENCE DETECTION EXPERIMENTS

A suite of experiments were created to test the theory of SR-replication and the user and Kernel mode implementation prototypes of SRRAT. The tests were conducted with viruses drawn from a collection of 445 virus samples. The collection was built from malware repositories on the Internet [60, 42]. The viruses in the collection were chosen to be representative of all the major categories of virus types. The amount of virus samples for each category is listed in Figure 5.1. All the sample viruses in the collection were scanned using Kaspersky Anti-Virus software [33] to validate their authenticity, name and classification. The focus of these tests was to count the total number of correct identifications of viruses plus the total amount of false negatives and false positives produced by the prototypes. All the tests were conducted on a desktop computer running Microsoft Windows XP with no anti-virus software installed. The testing involving viruses were done using Vmware Virtual Workstation [59], which allows for safe isolation of the viruses from infecting an actual machine while providing a rich real computer emulation environment.

Figure 5.1: Virus Classification with Total Samples amount

Virus Types	**Total Samples**
`Email Worms`	**110**
`Network Worms`	**99**
`Peer-to-Peer Worms`	**79**
`Instant Messaging Worms`	**6**
`Win32 Viruses`	**151**

5.1 Theory Validation

Testing the theory of SR-replication entails inquiring if this is a characteristic that occurs in multiple viruses across several different virus classifications and can further be identified in some manner. More importantly, it is pivotal to establish if SR-replication is a characteristic that does not occur in benign processes, this is one of my assumptions. Establishing these two points will indicate if SR-replication can be used to distinguish between viruses and benign processes and at the same time produce little or no false negatives and false positives.

My approach to test the theory of SR-replication was to execute several viruses and commonly used applications and operating system processes and have their Win32 API function calls and Kernel system service requests with source and destination arguments recorded and analyzed. The program used for this was API SPY 32 [7] which records Win32 API function calls and Process Monitor [55] which records Kernel system service requests. The benign processes used for testing are listed in Figures 5.1 and 5.2, these were chosen by logging all processes running on two computers on a 5 day span, the processes executed the most were chosen for testing. The viruses chosen for this test were randomly selected from the collection of 500 assuring that each category was represented in this test set. The 284 viruses used for testing are listed in Figure 5.2 and 5.2.

5.2 User Mode Prototype

In testing the user mode implementation prototype of SRRAT three criteria need to be analyzed, they are: false positive production, false negative production and usability as a real time monitor and detector. To test for false negative production a test set of viruses were executed one by one in the virtual machine software with

AcroRd32.exe	netbeans.exe
AcroRd32Info.exe	OUTLOOK.EXE
Ad-Aware.exe	pa.exe
AlbumDB2.exe	palaunch.exe
AsusProb.exe	pastatus.exe
bibtex.exe	pdflatex.exe
CFD.exe	PHOTOED.EXE
csrss.exe	POWERPNT.EXE
Deskup.exe	procexp.exe
devenv.exe	Procmon.exe
emule.exe	rundll32.exe
ErrorKiller.exe	services.exe
EXCEL.EXE	Skype.exe
Explorer.EXE	sol.exe
firefox.exe	sqlservr.exe

Figure 5.2: Theory Validation Test Benign Processes - 1

SRRAT running. False positive production was tested together with usability as a real time monitor and detector by running SRRAT on two actual computer desktops for three days under normal computer use. Both computers had full Internet access and carry heavy use of several popular desktop applications plus Internet programs. Installations of new software and updates to already existing software were purposely done during the testing period as well. Anti-virus software was present and running on both computers during testing. The viruses were chosen by using those that showed use of Win32 API function calls during their execution as recorded by the API SPY 32 log files. This resulted in a set of 66 viruses listed in Figure 5.3.

5.3 Kernel Mode Prototype

The Kernel Mode Prototype of SRRAT was tested using the same three criteria as that used for the user mode prototype: false positive production, false negative production and usability as a real time monitor and detector. Testing false positive production was conducted jointly with usability as a real time monitor and detector by executing SRRAT on two actual computer desktops for three days under normal computer use. The two computers had full Internet access and experience heavy daily use of many

FrameworkService.exe	svchost.exe
gbk2uni.exe	symlcsvc.exe
GoogleEarth.exe	SyncBackSE.exe
HWN.exe	System
IEXPLORE.EXE	TEXCNTR.EXE
iexplore.exe	TexFriend.exe
java.exe	tomcat5.exe
LimeWire.exe	verclsid.exe
MATLAB.exe	WCESCOMM.EXE
Mcshield.exe	WinEdt.exe
MemoryManagement.vshost.exe	winlogon.exe
MSACCESS.EXE	winmine.exe
mscorsvw.exe	WinRAR.exe
msnmsgr.exe	WINWORD.EXE
naPrdMgr.exe	wmiprvse.exe
nbexec.exe	wuauclt.exe

Figure 5.3: Theory Validation Test Benign Processes - 2

popular Internet and desktop applications. New software installations and updates to already existing software were purposely done during the testing period as well. Anti-virus software was running on both computers during testing. Testing for false negative production was done by executing a test set of viruses one at a time in the virtual machine software with SRRAT running. The viruses were chosen by using those that showed use of Kernel system services during their execution as recorded by the Process Monitor log files. This resulted in a set of 367 viruses listed in Figures 5.3, 5.3, 5.3 and 5.3.

Email Worms	**Peer to Peer Worms**	**Network Worms**	**Win32 Viruses**
Abotus	Abuva	3DStars	Aidlot
Actem	Adil	CodeGreen.a	Andras.7300
Agist.a	Alcan.a	Cycle.a	Apathy.5378
Alanis	AntiFizz	Ezio.a	Apoc.a
Aliz	Aplich	Francette.a	Arch.a
Altice	Apsiv	Francette.b	Aris
Amus.a	Aritim	Francette.c	Artelad.2173
Anarch	Ariver	Francette.d	Bacros.a
Android	Blaxe	Francette.e	Banaw.2157
Anel	Cabby	Francette.g	Barum.1536
Animan	Cake	Hiberium.b	Basket.a
Anpir.a	Carfin	Maslan.a	Bayan.a
Antiax	Cassidy	Maslan.b	BCB.a
Antites	Cayen.a	Mytob.q	Bee
Aplore	Cocker	Protoride.aa	Beef.2110
Apost	Compatex	Protoride.ai	Bender.1363
Assarm	Compux.a	Protoride.al	Bika.1906
Atirus	Cozit	Protoride.ar	Blateroz
Avoner	Dafly.b	Protoride.b	Bluback.1376
Babuin	Dani	Protoride.bk	Blueballs.4117
BabyBear	Delf.a	Protoride.e	Bogus.4096
Badass	Druagz	Protoride.f	Bondage.968.a
Badtrans.a	Agobot.a	Afire.b	Cabanas.e
Bagle.a	Agobot.b	Afire.c	DarkSide.1371
Bagle.j	Agobot.c	Afire.d	Elkern.a
Bagle.k	Agobot.d	Bozori.b	Enumiacs-6656
Bagle.m	Backterra.a	Bozori.e	Levi-2961
Bagle.n	Banuris.a	Bozori.j	Mental
Bagle.o	Bereb.a	Dabber.a	Mental-10000
Bandet.a	Erdam	Protoride.g	Butter
Banza	Flocker	Raleka.b	CabInfector
Bater.a	Franvir	Rega.a	Cecile
Benny	Furby	Salie.a	Civut.a
Bimoco.a	Gagse	SdBoter.a	Cloz.a
Black	Gotorm	SdBoter.b	Cmay.1222
Blare	Grompo	SdBoter.c	Cornad
Blitzy	Halfint	SdBoter.g	Crosser
Bonorm	Huntox	SdBoter.k	Delfer.a

Figure 5.4: Theory Validation Test Viruses - 1

Email Worms	**Peer to Peer Worms**	**Network Worms**	**Win32 Viruses**
Bormex	Ident	Shelp.a	Devir
Borzella	Insta.a	Spoder.a	Dictator.2304
Botter.a	Inter	Stap.b	Dislex
Bumper	Irkaz	Stap.e	Gipiras.a
Burnox	Kabak.a	Stap.f	Hezhi
Calil	Kamadina	Syner.a	Jlok
Calposa	Kamafe	Webdav.a	Kenfa.a
Carfrin	Kanyak.a	Welchia.a	Netlip
Cervivec	Kapucen.b	Welchia.b	Niya.a
CWS.a	Kazeus	Welchia.c	Porad.a
Dumaru.r	Bereb.b	Dabber.b	Neshta.a
Eyeveg.m	Gedza.b	Dabber.c	Parite.a
Happy	Kenfo	Welchia.e	Sinco
Klez.a	Gedza.c	Domwoot.c	Parite-b
Klez.e	Habaku.b	Doomjuice.b	Seppuku.6834
Klez.i	Kifie.a	Kidala.a	Small.a
Merkur.b	Kifie.c	Kidala.b	Small.b
Mimail.j	Kifie.f	Lebreat.a	Tapan-3882
Mydoom.ax	Niklas.b	Muma.b	Thorin.11932
Mydoom.b	Niklas.c	Muma.c	Thorin.b
Plexus.a	Opex.a	Opasoft.a	Thorin.c
Sircam.a	Polip.a	Padobot.m	Xorala
Sircam.d	Zaka.a	Sasser.b	Younga.4434
Sober.a	Zaka.f	Theals.c	ZMist
Sober.f	Zaka.m	Vesser.a	ZPerm.b
Yoxec	Kevor	Xatch.a	Spreder
Zar.a	Kovirz	Zan	Sugin
Zhangpo	Krepper	Zusha.a	TeddyBear
Zircon	Lamerx	Zusha.b	VChain
Zoek	Lemb.b	Zusha.c	Watcher.a
Zoher	Vagas.a	Zusha.e	Zevity
Zush	Walrain	Zusha.f	Zorg.a
Zwur.a	Weakas	Zusha.h	Zori.a

Figure 5.5: Theory Validation Test Viruses - 2

Email-Worm.Win32.Alanis	Net-Worm.Win32.Webdav.a
Email-Worm.Win32.Android	Net-Worm.Win32.Zusha.a
Email-Worm.Win32.Anpir.a	Net-Worm.Win32.Zusha.b
Email-Worm.Win32.Antiax	Net-Worm.Win32.Zusha.c
Email-Worm.Win32.Apost	Net-Worm.Win32.Zusha.e
Email-Worm.Win32.Asid.a	Net-Worm.Win32.Zusha.f
Email-Worm.Win32.Bandet.a	P2P-Worm.Win32.Agobot.a
Email-Worm.Win32.Bater.a	P2P-Worm.Win32.Agobot.b
Email-Worm.Win32.Benny	P2P-Worm.Win32.Agobot.c
Email-Worm.Win32.Bimoco.a	P2P-Worm.Win32.Agobot.d
Email-Worm.Win32.Bormex	P2P-Worm.Win32.Aplich
Email-Worm.Win32.Borzella	P2P-Worm.Win32.Blaxe
Email-Worm.Win32.Botter.a	P2P-Worm.Win32.Cassidy
Email-Worm.Win32.Burnox	P2P-Worm.Win32.Compux.a
Email-Worm.Win32.Calposa	P2P-Worm.Win32.Delf.a
Email-Worm.Win32.Canbis.a	P2P-Worm.Win32.Erdam
Email-Worm.Win32.Happy	P2P-Worm.Win32.Flocker.01
Email-Worm.Win32.Klez.b	P2P-Worm.Win32.Gagse
Email-Worm.Win32.Klez.c	P2P-Worm.Win32.Gedza.c
Email-Worm.Win32.Klez.d	P2P-Worm.Win32.Irkaz
Email-Worm.Win32.Klez.e	P2P-Worm.Win32.Kanyak.a
Email-Worm.Win32.Klez.f	P2P-Worm.Win32.Kapucen.b
Email-Worm.Win32.Klez.g	P2P-Worm.Win32.Weakas
Email-Worm.Win32.Klez.i	Virus.Win32.Arch.a
Email-Worm.Win32.Klez.j	Virus.Win32.BCB.a
Email-Worm.Win32.Sircam.d	Virus.Win32.Bee
Net-Worm.Win32.Doomran	Virus.Win32.Canbis.a
Net-Worm.Win32.Ezio.a	Virus.Win32.Jlok
Net-Worm.Win32.Maslan.b	Virus.Win32.Redemption
Net-Worm.Win32.Nimda	Virus.Win32.Small.c
Net-Worm.Win32.Rega.a	Virus.Win32.Spreder
Net-Worm.Win32.Sasser.b	Virus.Win32.Watcher.a
Net-Worm.Win32.Syner.a	Virus.Win32.Zori.a

Figure 5.6: Test Viruses for User implementation of SRRAT

V.W32.Zevity	V.W32.ZMist	V.W32.Zori.a
V.W32.ZPerm.b	V.W32.ZPerm.b2	V.Win9x.CIH
V.Win9x.DarkSide.1371	V.Win9x.Sledge.735.b	V.Win9x.Small.140
Worm.VB-16		

Figure 5.7: Test Viruses for Kernel implementation of SRRAT - 1

EW.3DStars	EW.W32.Bormex	EW.W32.NSky.d
EW.VBS.Homepage	EW.W32.Borzella	EW.W32.NSky.q
EW.VBS.Loveletter.A	EW.W32.Botter.a	EW.W32.Plexus-a
EW.W32.Actem	EW.W32.Bumper.a	EW.W32.Roach.b
EW.W32.Agist.a	EW.W32.Burnox	EW.W32.Sircam.a
EW.W32.Alanis	EW.W32.Antiman	EW.W32.Sircam.d
EW.W32.Amus.a	EW.W32.Calposa	EW.W32.Sober.a
EW.W32.Anarch	EW.W32.Canbis.a	EW.W32.Sober.f
EW.W32.Android	EW.W32.Carfrin	EW.W32.Sober.y
EW.W32.Anel	EW.W32.Cervivec	EW.W32.Xanax
EW.W32.Animan	EW.W32.CWS.a	EW.W32.Zafi.b
EW.W32.Anpir.a	EW.W32.Doombot.b	EW.W32.Zafi.d
EW.W32.Antiax	EW.W32.Dumaru.a	EW.W32.Zar.a
EW.W32.Antites	EW.W32.Dumaru.c	EW.W32.Zhangpo
EW.W32.Aplore	EW.W32.Dumaru.m	EW.W32.Zush
EW.W32.Apost	EW.W32.Dumaru.r	IM-Worm.W32.Bropia.aj
EW.W32.Appflet.a	EW.W32.Eyeveg.m	IM-Worm.W32.Aimes-b
EW.W32.Babuin.a	EW.W32.Klez.a	NW.W32.AllocUp-b
EW.W32.BabyBear.a	EW.W32.Klez.b	NW.W32.Afire.c
EW.W32.Baconex	EW.W32.Klez.c	NW.W32.Afire.d
EW.W32.Bagle.fj	EW.W32.Klez.d	NW.W32.BlueCode
EW.W32.Bagle.fk	EW.W32.Klez.e	NW.W32.Bobic.k
EW.W32.Bagle.h	EW.W32.Klez.f	NW.W32.Bozori.b
EW.W32.Bagle.i	EW.W32.Klez.g	NW.W32.Bozori.e
EW.W32.Bagle.j	EW.W32.Klez.i	NW.W32.Bozori.j
EW.W32.Bagle.k	EW.W32.Klez.j	NW.W32.CodeGreen.a
EW.W32.Bagle.m	EW.W32.LovGate.g	NW.W32.CodeRed
EW.W32.Bagle.n	EW.W32.Nyxem	NW.W32.Cycle.a
EW.W32.Bagle.o	EW.W32.Merkur.b	NW.W32.Dipnet.f
EW.W32.Bagle.q	EW.W32.Mimail.j	NW.W32.Dabber.b
EW.W32.Bandet.a	EW.W32.Mydoom.ax	NW.W32.Dabber.c
EW.W32.Banza	EW.W32.Mydoom.b	NW.W32.Daper.a
EW.W32.Bater.a	EW.W32.Mydoom.e	NW.W32.Domwoot.c
EW.W32.Benny	EW.W32.Mydoom.l	NW.W32.Doomjuice.b
EW.W32.Bimoco.a	EW.W32.Mydoom.m	NW.W32.Doomjuice.d
EW.W32.Blare	EW.W32.Mydoom.q	NW.W32.Doomran
EW.W32.Blitzy	EW.W32.NSky	NW.W32.Ezio.a
EW.W32.Bonorm	EW.W32.NSky.b	NW.W32.Francette.a
EW.W32.Asid.a	EW.W32.Happy	EW.W32.Klez
EW.W32.Atirus	V.W32.Xorala	V.W32.Xorala.b

Figure 5.8: Test Viruses for Kernel implementation of SRRAT - 2

NW.W32.Francette.b	NW.W32.SdBoter.k	P2PW.W32.Cabby
NW.W32.Francette.c	NW.W32.Shelp.a	P2PW.W32.Cake
NW.W32.Francette.d	NW.W32.Spoder.a	P2PW.W32.Carfin
NW.W32.Francette.e	NW.W32.Stap.b	P2PW.W32.Cassidy
NW.W32.Francette.g	NW.W32.Stap.e	P2PW.W32.Cayen.a
NW.W32.Hiberium.b	NW.W32.Stap.f	P2PW.W32.Cocker
NW.W32.Incef.a	NW.W32.Syner.a	P2PW.W32.Compatex
NW.W32.Incef.b	NW.W32.Theals.b	P2PW.W32.Compux.a
NW.W32.Kidala.a	NW.W32.Theals.c	P2PW.W32.Cozit
NW.W32.Kidala.b	NW.W32.Vesser.a	P2PW.W32.Dafly.b
NW.W32.Lebreat.a	NW.W32.Webdav.a	P2PW.W32.Dani
NW.W32.Lebreat.b	NW.W32.Welchia.a	P2PW.W32.Darby.b
NW.W32.Lebreat.d	NW.W32.Welchia.b	P2PW.W32.Darby.c
NW.W32.Lebreat.m	NW.W32.Welchia.c	P2PW.W32.Delf.a
NW.W32.Muma.c	NW.W32.Welchia.e	P2PW.W32.Disager
NW.W32.Maslan.b	NW.W32.Xatch.a	P2PW.W32.Druagz
NW.W32.Muma.b	NW.W32.Zan	P2PW.W32.Erdam
NW.W32.Muma.c	NW.W32.Zusha.a	P2PW.W32.Flocker.01
NW.W32.Mytob.q	NW.W32.Zusha.b	P2PW.W32.Franvir
NW.W32.Nimda	NW.W32.Zusha.c	P2PW.W32.Furby
NW.W32.OpasSoft.a.pac	NW.W32.Zusha.e	P2PW.W32.Gagse
NW.W32.Padobot.m	NW.W32.Zusha.f	P2PW.W32.Gedza.b
NW.W32.Protoride.aa	P2PW.W32.Abuva	P2PW.W32.Gedza.c
NW.W32.Protoride.ai	P2PW.W32.Adil	P2PW.W32.Gotorm
NW.W32.Protoride.al	P2PW.W32.Agobot.a	P2PW.W32.Grompo
NW.W32.Protoride.ar	P2PW.W32.Agobot.b	P2PW.W32.Habaku.a
NW.W32.Protoride.b	P2PW.W32.Agobot.c	P2PW.W32.Habku.b
NW.W32.Protoride.bk	P2PW.W32.Agobot.d	P2PW.W32.Halfint
NW.W32.Protoride.e	P2PW.W32.Alcan.a	P2PW.W32.Huntox
NW.W32.Protoride.f	P2PW.W32.AntiFizz	P2PW.W32.Ident
NW.W32.Protoride.g	P2PW.W32.Aplich	P2PW.W32.Ihit.a
NW.W32.Raleka.b	P2PW.W32.Apsiv	P2PW.W32.Insta.a
NW.W32.Rega.a	P2PW.W32.Aritim	P2PW.W32.Inter
NW.W32.Salie.a	P2PW.W32.Ariver	P2PW.W32.Irkaz
NW.W32.Sasser.b	P2PW.W32.Backterra.a	P2PW.W32.Kabak.a
NW.W32.Sasser.d	P2PW.W32.Banuris.a	P2PW.W32.Kamadina
NW.W32.SdBoter.a	P2PW.W32.Benjamin.a.exe	P2PW.W32.Kamafe
NW.W32.SdBoter.b	P2PW.W32.Bereb.a	P2PW.W32.Kanyak.a
NW.W32.SdBoter.c	P2PW.W32.Bereb.b	P2PW.W32.Kapucen.b
NW.W32.SdBoter.g	P2PW.W32.Blaxe	P2PW.W32.Kazeus

Figure 5.9: Test Viruses for Kernel implementation of SRRAT - 3

P2PW.W32.Kenfo	V.W32.Arch.a	V.W32.Gipiras.a
P2PW.W32.Kevor	V.W32.Aris	V.W32.Gpcode.ac
P2PW.W32.Kifie.a	V.W32.Artelad.2173	V.W32.Halen.2618
P2PW.W32.Kifie.c	V.W32.Bacros.a	V.W32.Hezhi
P2PW.W32.Kifie.f	V.W32.Banaw.2157	V.W32.Jlok
P2PW.W32.Lamerx	V.W32.Barum.1536	V.W32.Kenfa.a
P2PW.W32.Lemb.b	V.W32.Basket.a	V.W32.Levi.2961
P2PW.W32.Mantas.a	V.W32.Bayan.a	V.W32.Mental
P2PW.W32.Niklas.a	V.W32.BCB.a	V.W32.Mental.10000
P2PW.W32.Niklas.b	V.W32.Bee	V.W32.Mental.10472
P2PW.W32.Niklas.c	V.W32.Beef.2110	V.W32.Neshta.a
P2PW.W32.Opex.a	V.W32.Bender.1363	V.W32.Nlip
P2PW.W32.Polipos	V.W32.Bika.1906	V.W32.Niya.a
P2PW.W32.SpyBot	V.W32.Blateroz	V.W32.Parite.a
P2PW.W32.Vagas.a	V.W32.Bluback.1376	V.W32.Parite.b
P2PW.W32.Walrain	V.W32.Blueballs.4117	V.W32.Porad.a
P2PW.W32.Weakas	V.W32.Bogus.4096	V.W32.Redemption
P2PW.W32.Zaka.a	V.W32.Bondage.968.a	V.W32.Savior.1680
P2PW.W32.Zaka.f	V.W32.Butter	V.W32.Seppuku.6834
P2PW.W32.Zaka.m	V.W32.Cabanas.e	V.W32.Sinco
V.W32.Cloz.a	V.W32.CabInfector	V.W32.Small.a
V.W32.Storm-2	V.W32.Canbis.a	V.W32.Small.c
V.W32.Civut.a	V.W32.Cecile	V.W32.Spreder
V.Boot-DOS.Tequila	V.W32.Cmay.1222	V.W32.Stream.a
V.DOS.Maltese-Amoeba.2367	V.W32.Cornad	V.W32.Stream.b
V.DOS.OneHalf.3666	V.W32.Crosser	V.W32.Sugin
V.MSIL.Gastropod	V.W32.Crypto	V.W32.Tapan.3882
V.MSWord.Blaster	V.W32.CTX.6886	V.W32.TeddyBear
V.MSWord.Melissa	V.W32.Delfer.a	V.W32.Tenga.a
V.VBS.Lucky2	V.W32.Devir	V.W32.Teta.a
V.VBS.H	V.W32.Dictator.2304	V.W32.Thorin.11932
V.W32.Aidlot	V.W32.Dislex	V.W32.Thorin.b
V.W32.Aldebaran.8365.a	V.W32.Donut	V.W32.Thorin.c
V.W32.Aldebaran.8365.b	V.W32.Elkern.a	V.W32.Thorin.d
V.W32.Andras.7300	V.W32.Emotion.a	V.W32.Thorin.e
V.W32.Apathy.5378	V.W32.Enumiacs.6656	V.W32.VChain
V.W32.Apoc.a	V.W32.Fosforo	V.W32.Voltage.A
V.W32.Apparition	V.W32.Ghost.1667	V.W32.Watcher.a
V.W32.Storm	V.W32.Yerg.9571	V.W32.Younga.4434

Figure 5.10: Test Viruses for Kernel implementation of SRRAT - 4

CHAPTER 6

TEST RESULTS: ANALYSIS AND EVALUATION

Performing the tests was a long and strenuous process. The nature of virus testing requires several re-installations of the host computer to ensure a clean virus free environment for the next test. To ensure that each virus was executed in a virus free environment, the Vmware workstation virtual machine was restored to a clean state after concluding each test. Assuring a virus free environment for each test was needed to ensure that a virus was not kept form executing normally as a result of a previous virus's infection on the virtual machine. What follows is an analysis and evaluation of all the test results along with observations and experiences from conducting the tests.

6.1 Theory Validation

Conducting this test took approximately 4 days to complete. The benign process testing was completed in one day and the balance of days was taken by the virus testing. The test results for the benign processes are presented in Figure 6.1. The first and fourth columns are the names of each benign process tested, the second and fifth columns are the results of testing for SR, the third and sixth columns are the test results for SR-replication with Y meaning yes and N meaning No. When testing commenced I decided to also record any occurrence of SR. My reasoning for this was if a benign process was an SR process and did not attempt SR-replication during testing, the possibility of attempting SR-replication could still occur under different execution conditions. Therefore I considered an SR benign process as being a potential false positive assuming the correct execution conditions were in place for SR-replication to occur. The test results show that all 62 benign process not only did not attempt SR-replication but none even attempted SR. The result is the whole

test set can be classified as non-SR benign processes based on the test results. Not finding any SR-replication did not surprise me as this characteristic not being found in benign processes is one of my main assumptions in this research. I was surprised though that none of the processes attempted SR. As each process was executed I interacted with them in as many typical user ways as possible to afford maximum possibility to the process to exhibit different forms of behavior. Finding none of these processes attempted SR-replication and SR supports my assumption that SR can be used to distinguish between viral and non-viral processes. Furthermore the lack of SR reenforces my assumption by showing that not only do benign processes not attempt SR-replication but they may not even read themselves, thus not be an SR process, in any way during their execution. This further distinguishes benign from viral based on SR-replication characteristic and reduces the chances of false positive production.

All 284 viruses were tested one by one in the virtual machine for the attempt of SR-replication. The virtual machine was reset to a clean virus free state before each test was conducted. A summary of the virus results are in Figures 6.1. The full test results are in Figures 6.1, 6.1, 6.1 and 6.1. Analyzing the results it becomes clear that a majority of the viruses did in fact show SR-replication with the exception of the Win32 Virus class. For that class the majority, 58 viruses, did not show SR-replication. The viruses that did not show SR-replication could be the result of advanced anti-detection techniques. Some viruses have the capacity to detect running processes that may be used to terminate or erase them, if they detect such a process they will behave as a benign process and do nothing exhibiting virus like behavior, this of course includes replication. Another reason for these viruses not showing SR-replication is they may have not found the right conditions to replicate. Win32 viruses tend to infect files that are of a specific format, most notably the Portable Execution

Benign Process	SR	SRR	Benign Process	SR	SRR
AcroRd32.exe	N	N	netbeans.exe	N	N
AcroRd32Info.exe	N	N	OUTLOOK.EXE	N	N
Ad-Aware.exe	N	N	pa.exe	N	N
AlbumDB2.exe	N	N	palaunch.exe	N	N
AsusProb.exe	N	N	pastatus.exe	N	N
bibtex.exe	N	N	pdflatex.exe	N	N
CFD.exe	N	N	PHOTOED.EXE	N	N
csrss.exe	N	N	POWERPNT.EXE	N	N
Deskup.exe	N	N	procexp.exe	N	N
devenv.exe	N	N	Procmon.exe	N	N
emule.exe	N	N	rundll32.exe	N	N
ErrorKiller.exe	N	N	services.exe	N	N
EXCEL.EXE	N	N	Skype.exe	N	N
Explorer.EXE	N	N	sol.exe	N	N
firefox.exe	N	N	sqlservr.exe	N	N
FrameworkService.exe	N	N	svchost.exe	N	N
gbk2uni.exe	N	N	symlcsvc.exe	N	N
GoogleEarth.exe	N	N	SyncBackSE.exe	N	N
HWN.exe	N	N	System	N	N
IEXPLORE.EXE	N	N	TEXCNTR.EXE	N	N
iexplore.exe	N	N	TexFriend.exe	N	N
java.exe	N	N	tomcat5.exe	N	N
LimeWire.exe	N	N	verclsid.exe	N	N
MATLAB.exe	N	N	WCESCOMM.EXE	N	N
Mcshield.exe	N	N	WinEdt.exe	N	N
MemoryManagement.vshost.exe	N	N	winlogon.exe	N	N
MSACCESS.EXE	N	N	winmine.exe	N	N
mscorsvw.exe	N	N	WinRAR.exe	N	N
msnmsgr.exe	N	N	WINWORD.EXE	N	N
naPrdMgr.exe	N	N	wmiprvse.exe	N	N
nbexec.exe	N	N	wuauclt.exe	N	N

Figure 6.1: Theory Validation Test Results Benign Processes

	Email Worms	Peer-to-Peer Worms	Network Worms	Win32 Viruses
SR-replication	43	47	45	13
No SRreplication	28	24	26	58

Figure 6.2: Summary Results Theory Validation Virus Test

(PE) format. It is possible these viruses searched for victim files and simply did not find any and thus could not replicated. A second interesting observation from the results is where the SR-replication occurred. Of the viruses that did replicate, the overwhelming majority did so in Kernel mode and a smaller amount replicated in user mode. Only three viruses replicated in both user and Kernel mode. The implication of the majority of these viruses replicating in Kernel is they do this purposely to attempt detection avoidance. By executing in Kernel mode they have the capacity to run below or at the same level as virus detectors thus allowing them more leeway to hide and avoid detection. Just considering only the static analysis, the viruses that did not show SR-replication are false negatives. It is however difficult to say if they really could be false negatives for the reasons stated here, it is possible they could be detected with the proper virus detector in place.

Overall the theory validation testing results strongly support my assumption that SR-replication can distinguish between viruses and benign processes. The key to this conclusion is the fact that no false positives occurred and several true positives occurred. If false positives had occurred then one can conclude that SR-replication is a characteristic generally occurring in any process. The lack of SR-replication and SR itself in the benign processes suggests the opposite, that SR-replication may in fact be a characteristic unique to viruses and not occurring in benign processes.

Email Worms	*SRR* Attempts	API Call	Kernel Service	Peer to Peer Worms	*SRR* Attempts	API Call	Kernel Service
Abotus	0	N	N	Abuva	0	N	N
Actem	0	N	N	Adil	3	Y	Y
Agist.a	1	N	Y	Alcan.a	1	N	Y
Alanis	7	N	Y	AntiFizz	1	N	Y
Aliz	0	N	N	Aplich	1	N	Y
Altice	0	N	N	Apsiv	0	N	N
Amus.a	10	N	Y	Aritim	1	N	Y
Anarch	1	N	Y	Ariver	1	N	Y
Android	1	Y	N	Blaxe	6	N	Y
Anel	0	N	N	Cabby	1	N	Y
Animan	1	N	Y	Cake	0	N	N
Anpir.a	4	Y	N	Carfin	0	N	N
Antiax	1	Y	N	Cassidy	19	Y	N
Antites	0	N	N	Cayen.a	0	N	N
Aplore	2	N	Y	Cocker	61	N	Y
Apost	1	N	Y	Compatex	6	N	Y
Assarm	1	Y	N	Compux.a	36	Y	Y
Atirus	1	N	Y	Cozit	2	N	Y
Avoner	0	N	N	Dafly.b	0	N	N
Babuin	1	N	Y	Dani	0	N	N
BabyBear	0	N	N	Delf.a	1	Y	N
Badass	0	N	N	Druagz	1	N	Y
Badtrans.a	0	N	N	Agobot.a	1	Y	N
Bagle.a	1	N	Y	Agobot.b	1	Y	N
Bagle.j	1	N	Y	Agobot.c	1	Y	N
Bagle.k	1	N	Y	Agobot.d	1	Y	N
Bagle.m	1	N	Y	Backterra.a	0	N	N
Bagle.n	1	N	Y	Banuris.a	217	N	Y
Bagle.o	1	N	Y	Bereb.a	474	N	Y
Bandet.a	2	Y	N	Erdam	1	Y	N
Banza	0	N	N	Flocker	1	Y	N
Bater.a	1	Y	N	Franvir	0	N	N
Benny	1	N	Y	Furby	0	N	N
Bimoco.a	1	Y	N	Gagse	257	N	Y
Black	0	N	N	Gotorm	1	N	Y
Blare	0	N	N	Grompo	1	N	Y
Blitzy	0	N	N	Halfint	0	N	N
Bonorm	0	N	N	Huntox	0	N	N

Figure 6.3: Theory Validation Test Results Viruses - 1

Email Worms	SRR Attempts	API Call	Kernel Service	Peer to Peer Worms	SRR Attempts	API Call	Kernel Service
Bormex	0	N	N	Ident	0	N	N
Borzella	1	Y	N	Insta.a	7	N	Y
Botter.a	1	N	Y	Inter	1	N	Y
Bumper	0	N	N	Irkaz	2	Y	N
Burnox	30	N	Y	Kabak.a	0	N	N
Calil	0	N	N	Kamadina	0	N	N
Calposa	8	N	Y	Kamafe	0	N	N
Carfrin	1	N	Y	Kanyak.a	1	Y	N
Cervivec	1	N	Y	Kapucen.b	1	Y	N
CWS.a	0	N	N	Kazeus	1	N	Y
Dumaru.r	3	N	Y	Bereb.b	481	N	Y
Eyeveg.m	1	N	Y	Gedza.b	0	N	N
Happy	0	N	Y	Kenfo	0	N	N
Klez.a	3	Y	Y	Gedza.c	2	N	Y
Klez.e	1	Y	N	Habaku.b	0	N	N
Klez.i	1	Y	N	Kifie.a	2	N	Y
Merkur.b	0	N	N	Kifie.c	2	N	Y
Mimail.j	1	N	Y	Kifie.f	3	N	Y
Mydoom.ax	1	N	Y	Niklas.b	2	N	Y
Mydoom.b	1	N	Y	Niklas.c	2	N	Y
Plexus.a	1	N	Y	Opex.a	103	N	Y
Sircam.a	0	N	N	Polip.a	0	N	N
Sircam.d	1	Y	N	Zaka.a	1	N	Y
Sober.a	2	N	Y	Zaka.f	1	N	Y
Sober.f	3	N	Y	Zaka.m	1	N	Y
Yoxec	0	N	N	Kevor	0	N	N
Zar.a	3	N	Y	Kovirz	0	N	N
Zhangpo	1	N	Y	Krepper	0	N	N
Zircon	0	N	N	Lamerx	0	N	N
Zoek	0	N	N	Lemb.b	3	N	Y
Zoher	0	N	N	Vagas.a	4	N	Y
Zush	0	N	N	Walrain	30	N	Y
Zwur.a	0	N	N	Weakas	1	Y	N

Figure 6.4: Theory Validation Test Results Viruses - 2

Network Worms	*SRR* Attempts	API Call	Kernel Service	Win32 Viruses	*SRR* Attempts	API Call	Kernel Service
3DStars	0	N	N	Aidlot	0	N	N
CodeGreen.a	0	N	N	Andras.7300	0	N	N
Cycle.a	1	N	Y	Apathy.5378	1	N	Y
Ezio.a	1	Y	N	Apoc.a	0	N	N
Francette.a	0	N	N	Arch.a	1	N	Y
Francette.b	0	N	N	Aris	0	N	N
Francette.c	0	N	N	Artelad.2173	0	N	N
Francette.d	0	N	N	Bacros.a	4	N	Y
Francette.e	0	N	N	Banaw.2157	0	N	N
Francette.g	0	N	N	Barum.1536	0	N	N
Hiberium.b	0	N	N	Basket.a	0	N	N
Maslan.a	1	N	Y	Bayan.a	0	N	N
Maslan.b	1	N	Y	BCB.a	60	N	Y
Mytob.q	1	N	Y	Bee	2	N	Y
Protoride.aa	0	N	N	Beef.2110	0	N	N
Protoride.ai	0	N	N	Bender.1363	0	N	N
Protoride.al	0	N	N	Bika.1906	0	N	N
Protoride.ar	0	N	N	Blateroz	0	N	N
Protoride.b	0	N	N	Bluback.1376	0	N	N
Protoride.bk	1	N	Y	Blueballs.4117	0	N	N
Protoride.e	0	N	N	Bogus.4096	0	N	N
Protoride.f	0	N	N	Bondage.968.a	0	N	N
Afire.b	3	N	Y	Cabanas.e	0	N	N
Afire.c	1	N	Y	DarkSide.1371	0	N	N
Afire.d	1	N	Y	Elkern.a	0	N	N
Bozori.b	1	N	Y	Enumiacs-6656	0	N	N
Bozori.e	1	N	Y	Levi-2961	0	N	N
Bozori.j	1	N	Y	Mental	0	N	N
Dabber.a	1	N	Y	Mental-10000	0	N	N
Protoride.g	0	N	N	Butter	0	N	N
Raleka.b	0	N	N	CabInfector	0	N	N
Rega.a	2	Y	N	Cecile	0	N	N
Salie.a	0	N	N	Civut.a	1	N	Y
SdBoter.a	1	N	Y	Cloz.a	0	N	N
SdBoter.b	1	N	Y	Cmay.1222	0	N	N
SdBoter.c	1	N	Y	Cornad	1	Y	N
SdBoter.g	1	N	Y	Crosser	0	N	N
SdBoter.k	1	N	Y	Delfer.a	0	N	N

Figure 6.5: Theory Validation Test Results Viruses - 3

Network Worms	SRR Attempts	API Call	Kernel Service	Win32 Viruses	SRR Attempts	API Call	Kernel Service
Shelp.a	0	N	N	Devir	0	N	N
Spoder.a	0	N	N	Dictator.2304	0	N	N
Stap.b	8	N	Y	Dislex	0	N	N
Stap.e	7	N	Y	Gipiras.a	0	N	N
Stap.f	7	N	Y	Hezhi	0	N	N
Syner.a	1	Y	N	Jlok	2	Y	Y
Webdav.a	9	Y	N	Kenfa.a	0	N	N
Welchia.a	1	N	Y	Netlip	0	N	N
Welchia.b	1	N	Y	Niya.a	0	N	N
Welchia.c	1	N	Y	Porad.a	0	N	N
Dabber.b	1	N	Y	Neshta.a	0	N	N
Dabber.c	1	N	Y	Parite.a	1	N	Y
Welchia.e	1	N	Y	Sinco	0	N	N
Domwoot.c	1	N	Y	Parite-b	1	N	Y
Doomjuice.b	1	N	Y	Seppuku.6834	0	N	N
Kidala.a	1	N	Y	Small.a	0	N	N
Kidala.b	1	N	Y	Small.b	0	N	N
Lebreat.a	2	N	Y	Tapan-3882	0	N	N
Muma.b	1	N	Y	Thorin.11932	0	N	N
Muma.c	1	N	Y	Thorin.b	0	N	N
Opasoft.a	0	N	N	Thorin.c	0	N	N
Padobot.m	1	N	Y	Xorala	0	N	N
Sasser.b	1	Y	N	Younga.4434	0	N	N
Theals.c	0	N	N	ZMist	1	N	Y
Vesser.a	0	N	N	ZPerm.b	0	N	N
Xatch.a	0	N	N	Spreder	0	N	N
Zan	0	N	N	Sugin	0	N	N
Zusha.a	1	Y	N	TeddyBear	0	N	N
Zusha.b	1	N	Y	VChain	0	N	N
Zusha.c	1	N	Y	Watcher.a	1	Y	N
Zusha.e	1	N	Y	Zevity	0	N	N
Zusha.f	1	N	Y	Zorg.a	0	N	N
Zusha.h	1	N	Y	Zori.a	1	Y	N

Figure 6.6: Theory Validation Test Results Viruses - 4

6.2 User Mode Prototype

Testing the user implementation of SRRAT against the 66 test viruses was conducted in less than a day. The detection of SR-replication for the viruses is listed in Figure 6.2. Out of 66 viruses in the test set 18 were terminated and flagged as attempting to execute SR-replication. When each of these viruses were terminated by SRRAT, the virus's SR-replication graph was created and saved to a text file. The SR-replication graph for the Alanis email worm is presented in Figure 6.7. The graph shows the Alanis worm attempted SR-replication by first invokng the `Readfile` API function with itself as the source parameter, the function returned the memory address 1568460 pointing to the buffer containing the read portion of the virus, this function call makes Alanis an SR process for invoking a read general operation using itself as the source of the read, thus Alanis is reading itself. The virus then called the `Writefile` Api function using the memory address 1568460 as the source of the write and the destination was the file kerneldll32.api. When this function was called SRRAT established transitivity between kerneldll32.api and gallo.exe which is the virus file itself. This positive test for transitivity showed Alanis to be attempting SR-replication and was terminated. SRRAT always terminated these processes before the actual `Writefile` function was invoked, this prevented the SR-replication from completing. Furthermore the graph show the read operation was the first operation to occur dealing with SR, this is noted by the 1 next to the function name, the number 2 next to the write operation function name indicates this operation was then the second that occurred dealing with SR. The significance of this numbering is that SRRAT not only terminated Alanis for attempting SR-replication but it terminated Alanis on it's first attempt of SR-replication.

I classified the viruses that were not terminated into two groups: those viruses not hooked by SSRAT listed in Figure 6.2 and those viruses that did not attempt SR-

```
NODE: C:\Documents and Settings\JAM-VX-
MACHINE\Desktop\gallo.exe -1
  NODE--Out--> ReadFile  1  read  NULL 1564680
========================================
NODE: NULL 1564680
  NODE--Out--> WriteFile  2  write  C:\WINDOWS\
system32\kerneldll32.api -1
========================================
NODE: C:\WINDOWS\system32\kerneldll32.api -1
========================================
+ENDofGRAPH+
```

Figure 6.7: EW-Win32.Alanis SR-replication graph

replication during testing which are listed in Figure 6.2. Of the remaining 48 viruses that were not terminated, 15 of them executed and did not attempt SR-replication by the use of API function calls in a way that was detectable by SRRAT. Some of these viruses perform SR-replication in Kernel mode and others will only replicate when certain conditions are met and quite possibly these conditions were not present in the virtual machine. Interestingly, 5 of these viruses: `watcher.a`, `weakas`, `rega.a`, `delf.a` and `ezio.a` had previously attempted SR-replication during the theory validation testing. During that testing the SR-replication had been identified by the log files of API SPY 32. I later concluded that these 5 viruses that should have been detected were not as a result of the implementation of SRRAT missing some key functionality which prevented detection from occurring.

Of the 48 viruses not terminated by SRRAT, 33 were not hooked by SRRAT when execution commenced. SRRAT notifies me through a log file of it's activities while it runs. When it hooks a process the action is noted in the log file. When each of the 33 viruses listed in Figure 6.2 were executed one by one, the SRRAT log file did

Virus Name	*SRR* Detected	Virus Name	*SRR* Detected
Email-Worm.Win32.Alanis	Y	Net-Worm.Win32.Webdav.a	N
Email-Worm.Win32.Android	N	Net-Worm.Win32.Zusha.a	N
Email-Worm.Win32.Anpir.a	N	Net-Worm.Win32.Zusha.b	N
Email-Worm.Win32.Antiax	N	Net-Worm.Win32.Zusha.c	N
Email-Worm.Win32.Apost	Y	Net-Worm.Win32.Zusha.e	Y
Email-Worm.Win32.Asid.a	N	Net-Worm.Win32.Zusha.f	Y
Email-Worm.Win32.Bandet.a	N	P2P-Worm.Win32.Agobot.a	Y
Email-Worm.Win32.Bater.a	N	P2P-Worm.Win32.Agobot.b	Y
Email-Worm.Win32.Benny	N	P2P-Worm.Win32.Agobot.c	Y
Email-Worm.Win32.Bimoco.a	N	P2P-Worm.Win32.Agobot.d	Y
Email-Worm.Win32.Bormex	N	P2P-Worm.Win32.Aplich	N
Email-Worm.Win32.Borzella	Y	P2P-Worm.Win32.Blaxe	Y
Email-Worm.Win32.Botter.a	N	P2P-Worm.Win32.Cassidy	Y
Email-Worm.Win32.Burnox	Y	P2P-Worm.Win32.Compux.a	N
Email-Worm.Win32.Calposa	Y	P2P-Worm.Win32.Delf.a	N
Email-Worm.Win32.Canbis.a	N	P2P-Worm.Win32.Erdam	N
Email-Worm.Win32.Happy	N	P2P-Worm.Win32.Flocker.01	Y
Email-Worm.Win32.Klez.b	N	P2P-Worm.Win32.Gagse	Y
Email-Worm.Win32.Klez.c	N	P2P-Worm.Win32.Gedza.c	N
Email-Worm.Win32.Klez.d	N	P2P-Worm.Win32.Irkaz	N
Email-Worm.Win32.Klez.e	N	P2P-Worm.Win32.Kanyak.a	N
Email-Worm.Win32.Klez.f	N	P2P-Worm.Win32.Kapucen.b	Y
Email-Worm.Win32.Klez.g	N	P2P-Worm.Win32.Weakas	N
Email-Worm.Win32.Klez.i	N	Virus.Win32.Arch.a	N
Email-Worm.Win32.Klez.j	N	Virus.Win32.BCB.a	Y
Email-Worm.Win32.Sircam.d	N	Virus.Win32.Bee	N
Net-Worm.Win32.Doomran	N	Virus.Win32.Canbis.a	N
Net-Worm.Win32.Ezio.a	N	Virus.Win32.Jlok	N
Net-Worm.Win32.Maslan.b	N	Virus.Win32.Redemption	Y
Net-Worm.Win32.Nimda	N	Virus.Win32.Small.c	N
Net-Worm.Win32.Rega.a	N	Virus.Win32.Spreder	N
Net-Worm.Win32.Sasser.b	N	Virus.Win32.Watcher.a	N
Net-Worm.Win32.Syner.a	N	Virus.Win32.Zori.a	N

Figure 6.8: Virus Test Results User implementation of SRRAT

Virus Name	Virus Name
Email-Worm.Win32.Android	Email-Worm.Win32.Anpir.a
Email-Worm.Win32.Antiax	Email-Worm.Win32.Asid.a
Email-Worm.Win32.Bandet.a	Email-Worm.Win32.Bater.a
Email-Worm.Win32.Benny	Email-Worm.Win32.Bimoco.a
Email-Worm.Win32.Bormex	Email-Worm.Win32.Canbis.a
Email-Worm.Win32.Klez.b	Email-Worm.Win32.Klez.c
Email-Worm.Win32.Klez.d	Email-Worm.Win32.Klez.e
Email-Worm.Win32.Klez.f	Email-Worm.Win32.Klez.g
Email-Worm.Win32.Klez.i	Email-Worm.Win32.Klez.j
Email-Worm.Win32.Sircam.d	Net-Worm.Win32.Maslan.b
Net-Worm.Win32.Nimda	Net-Worm.Win32.Sasser.b
Net-Worm.Win32.Syner.a	Net-Worm.Win32.Webdav.a
P2P-Worm.Win32.Compux.a	P2P-Worm.Win32.Erdam
P2P-Worm.Win32.Gedza.c	P2P-Worm.Win32.Irkaz
P2P-Worm.Win32.Kanyak.a	Virus.Win32.Bee
Virus.Win32.Jlok	Virus.Win32.Small.c
Virus.Win32.Zori.a	

Figure 6.9: Viruses not hooked by User implementation of SRRAT

Virus Name	Virus Name	Virus Name
Virus.Win32.Watcher.a	Virus.Win32.Spreder	Virus.Win32.Canbis.a
Virus.Win32.Arch.a	P2P-Worm.Win32.Weakas	P2P-Worm.Win32.Delf.a
P2P-Worm.Win32.Aplich	Net-Worm.Win32.Zusha.c	Net-Worm.Win32.Zusha.b
Net-Worm.Win32.Zusha.a	Net-Worm.Win32.Rega.a	Net-Worm.Win32.Ezio.a
Net-Worm.Win32.Doomran	Email-Worm.Win32.Happy	Email-Worm.Win32.Botter.a

Figure 6.10: Viruses not Exhibiting SR-replication in User Mode SRRAT Testing

not contain any entry documenting a successful hook of the executing virus. These viruses executed fully on the system with no monitoring of them being conducted by SRRAT. Some of these viruses actually run in Kernel mode and are able to bypass user mode detectors such as SSRAT. But others do show usage of API function calls in user mode. These were not detected due to lack of functionality in the user mode implementation of SRRAT.

Testing for false positives occurred together with usability as a real time monitor by running SRRAT on two desktop computers for three days. During this time the two computers were used under normal conditions plus some installation programs were purposely run in an attempt to cause SRRAT to produce a false positive. At the end of the three days SRRAT did not report a single process as having attempted SR-replication, no processes were terminated as a result of exhibiting possible virus behavior which ultimately means that no false positives were produced. The testing also showed the user mode implementation of SRRAT not to be a very practical real time monitor and detector. One five occasions one of the computers had to be rebooted due to very slow operation resulting from SRRAT consuming high amounts of resources thus starving all the other processes running on the computer. On several occasions, SRRAT would crash when attempting to hook a process that was running at the time SRRAT was started. On a few occasions when SRRAT was terminated it still kept running and the process had to be terminated directly and ungracefully using Windows system tools. These problems were all implementation related and despite them no false positives occur and virus detection had been successful in some cases.

Overall, I feel the testing of the user mode implementation of SRRAT had mixed results. On the one hand detecting a subset of the test viruses shows that detection of

SR-replication in user mode is possible. The non-production of false positives further reenforces the idea that SR-replication is a characteristic unique to viruses. On the other hand, implementation issues due to lack of programming knowledge within the Windows environment may have led to some false negative production and a resource intensive implementation causing many problems that made it not to be the best choice as a practical tool for real time monitoring and detection of SR-replication in currently running processes. Only with increased programming experience in this area can a leaner, more robust and effective implementation tool be built.

6.3 Kernel Mode Prototype

A total of 14 days was need to test the Kernel mode implementation of SRRAT against the 367 test viruses and false positive testing. To test each virus required 8 days with the balance of days being used for false positive and usability testing. A summary of the test results is listed in Figure 6.3. As we can see from the summary the overall testing result showed half of the test viruses to exhibit SR-replication behavior with the other have not exhibiting this behavior.

Recall the memory problems encountered during creation of the implementation were not overcome and these results were built from analysis of the log files produced by the Kernel Mode implementation of SRRAT. Besides the four main categories of viruses I also added one and two samples of two new categories which were Instant Messaging viruses and Win32 Worms. These are not major categories of my test set and they were added just to have at least one sample to make the test set representative of other virus categories.

Viewing the results by virus category is is clear that SR-replication occured in the majority of viruses in the categories of: email worms, network worms, peer-to-peer

worms, instant messaging worms and Win32 worms. The main cause of the 50/50 split in the overall results is directly related to the very high false negative rate produced by the Win32 viruses category.

	Email Worms	Peer-to-Peer Worms	Network Worms	IM Worms	Win32 Worms	Win32 Viruses	Total Amount
SR-replication	67	50	45	2	1	19	184
No SRreplication	28	28	38	0	0	89	183
True Positve	70%	64%	54%	100%	100%	18%	50%
False Negative	30%	36%	46%	0%	0%	82%	50%

Figure 6.11: Summary Results Kernel Implementation SRRAT Virus Test

The viruses showing SR replication did so in one of two basic forms. The first form was a simple read and write general operations. This form was not the dominant one in the log file analysis of the virus executions. A sample of this form is in Figure 6.12. The second form and by far the most dominant was a sequence of operations that began with reading a file into memory followed by another reading of that memory to a new memory location and finally writing the memory to a new file. A sample of this form is in Figure 6.13. As seen from the test results in Figures 6.1, 6.1, 6.1 and 6.1, several of the viruses that attempted SR-replication did so multiple times, the same was true in this testing. The log files clearly showed multiple attempts to perform SR-replication by several of the test viruses.

The Win32 viruses which produced the highest number of false negatives, were for the most part the same viruses used in the testing in Figure 6.1 and 6.1. In that testing these viruses showed no attempts whatsoever of SR-replication. In testing these viruses again with the Kernel implementation of SRRAT those results were confirmed by the log file analysis. As it turns out by studying the log files, these viruses either: 1. make a copy of the virus itself into memory one or more times. In many cases this copy into memory is into the memory space of a currently running

```
00013263    NewZwReadFile ProcessName: ew-win32-Amus-a.exe
00013264    DestMemory: 1581032
00013265    Size: 65024
00013269    NewZwReadFile called!.
00013270    SelfReference Detected
00013271       File Path: \Documents and Settings\JAM-VX-
MACHINE\Desktop\ew-win32-Amus-a.exe
00013272       Bytes read: 65024
00013273    NewZwWriteFile ProcessName: ew-win32-Amus-a.exe
00013274    SourceMemory: 1581032
00013275    DestFile: \Documents and Settings\JAM-VX-
MACHINE\Desktop\ew-win32-Amus-a.exe
00013276    Size: 51782
00013283    NewZwWriteFile called!.
00013284      File Path: \WINDOWS\KdzEregli.exe
00013285      Bytes written: 51782
```

Figure 6.12: SRRAT Kernel Mode Log File Amus Virus

```
00004801    NewZwCreateFile ProcessName: ew-win32-Borzella.exe
00004803    Filepath: \??\C:\Documents and Settings\JAM-VX-
MACHINE\Desktop\ew-win32-Borzella.exe
00004804    Desired Access:
00004805       GENERIC_READ
00004814    FileHandle: 56
00005031    NewZwCreateSection ProcessName: ew-win32-Borzella.exe
00005032    NewZwCreateSection FileHandle: 56
00005033    NewZwCreateSection OUT SectionHandle: 64
00005036    NewZwMapViewOfSection ProcessName: ew-win32-Borzella.exe
00005038    NewZwMapViewOfSection SectionHandle: 64
00005039    NewZwMapViewOfSection ProcessHandle: -1
00005046    After call ------------------------
00005047    NewZwMapViewOfSection BaseAddress: 13565952
00005053    NewZwWriteFile ProcessName: ew-win32-Borzella.exe
00005054    SourceMemory: 13565952
00005055    DestFile:
00005128       File Path: \WINDOWS\dllmgr.exe
```

Figure 6.13: SRRAT Kernel Mode Log File Borzella Virus

process. or 2. did not attempt to replicate in any fashion at all. This can be the result of the failure to find a suitable environment or victim file to replicate. Given that these viruses performed poorly during the theory validation testing it is not at all surprising those findings would be confirmed here as well.

Excluding the Win32 viruses category, the rest of the false negatives produced in the other categories result from none of their log files showing any attempt to executed SR-replication. In several cases these viruses did copy the virus itself into the memory of currently running processes. Interestingly there were a few viruses that never attempted replication at all. These viruses I consider false negatives as well because their lack of replication can be from the absence of a suitable environment needed to replicate. These viruses may in fact replicate and may even perform SR-replication given the environment facilitating this for each virus.

False positive testing along with testing for usability as a real time monitor of the Kernel implementation of SRRAT was conducted across 4 days. The log files produced by SRRAT were saved once per hour and were analyzed when the testing was completed. Analyzing the log files showed no attempts SR-replication by any of the processes recorded. Furthermore no SR operations were conducted either by any process. This gives further support to my assumption of SR-replication being a characteristic unique to viruses. From a usability standpoint this version is very robust not causing and crashing or slowdown of the system at any point during testing. Furthermore it was never disabled or terminated by any virus during testing.

Overall I was quite happy with the testing results of the Kernel implementation of SRRAT. The number of true positives was much higher than those produced by the user implementation of SRRAT and no false positives occurred. The one disap-

pointment though not surprising was the high false negative amount of the Win32 viruses category. The Kernel implementation of SRRAT proved to be superior to the user mode in many aspects. It ran leaner, more robust, never crashed or slowed down the system at any time and proved capable of detecting far more viruses exhibiting SR-replication attempts then its user mode counterpart. Given this version is more capable of true positive detection then the user implementation version along with an overall 50% false negative production indicates to me this approach may be best used in conjunction with other known approaches to compensate their detection abilities with the false negatives produced by this implementation. The complete results of the virus testing with the Kernel mode version of SRRAT are listed in Figures 6.4, 6.4, 6.4, 6.4 and 6.4.

6.4 Evaluation of Proposed Solution

Analyzing the results of all the testing two conclusions can be made about SR-replication. First it seems clear that this form of replication is unique to viruses and not to benign processes. It may therefore be suitable as a characteristic to differentiate between the viruses and benign processes. Second, implementing this theory is better suited at the lowest possible level of a system to maximize detection capabilities. This is evident from the much larger number of true positives produced by the kernel mode of SRRAT then the user mode of SRRAT.

The false negative production can be from one of two observations each with its own unique solution: First the viruses replicate at different levels from those in my implementations or they are able to avoid detection, this would require better programming techniques which is realizable. Second, these viruses may in fact replicate and my implementations simply lacks some functionality to detect these viruses and this functionality is not implementable. In this case, the best solution would be to

compliment this approach with other known approaches with the assumption that the combination will reduce the false negatives while at the same time maintain or increase the true positives.

Virus Name	*SRR* Detected	Virus Name	*SRR* Detected
EW.3DStars	N	EW.W32.Bormex	N
EW.W32.NSky.d	Y	EW.VBS.Homepage	N
EW.W32.Borzella	Y	EW.W32.NSky.q	N
EW.VBS.Loveletter.A	N	EW.W32.Botter.a	Y
EW.W32.Plexus-a	Y	EW.W32.Actem	N
EW.W32.Bumper.a	N	EW.W32.Roach.b	Y
EW.W32.Agist.a	Y	EW.W32.Burnox	Y
EW.W32.Sircam.a	N	EW.W32.Alanis	N
EW.W32.Antiman	Y	EW.W32.Sircam.d	Y
EW.W32.Amus.a	Y	EW.W32.Calposa	Y
EW.W32.Sober.a	N	EW.W32.Anarch	Y
EW.W32.Canbis.a	Y	EW.W32.Sober.f	N
EW.W32.Android	Y	EW.W32.Carfrin	Y
EW.W32.Sober.y	N	EW.W32.Anel	N
EW.W32.Cervivec	Y	EW.W32.Xanax	Y
EW.W32.Animan	Y	EW.W32.CWS.a	N
EW.W32.Zafi.b	Y	EW.W32.Anpir.a	Y
EW.W32.Doombot.b	Y	EW.W32.Zafi.d	Y
EW.W32.Antiax	Y	EW.W32.Dumaru.a	Y
EW.W32.Zar.a	Y	EW.W32.Antites	N
EW.W32.Dumaru.c	Y	EW.W32.Zhangpo	N
EW.W32.Aplore	Y	EW.W32.Dumaru.m	Y
EW.W32.Zush	N	EW.W32.Apost	Y
EW.W32.Dumaru.r	Y	IM-Worm.W32.Bropia.aj	Y
EW.W32.Appflet.a	Y	EW.W32.Eyeveg.m	Y
IM-Worm.W32.Aimes-b	Y	EW.W32.Asid.a	Y
EW.W32.Happy	Y	EW.W32.Babuin.a	Y
EW.W32.Atirus	Y	EW.W32.Klez	N
EW.W32.Klez.a	N	NW.W32.AllocUp-b	Y
EW.W32.BabyBear.a	N	EW.W32.Klez.b	N
NW.W32.Afire.c	Y	EW.W32.Baconex	Y
EW.W32.Klez.c	Y	NW.W32.Afire.d	N
EW.W32.Bagle.fj	Y	EW.W32.Klez.d	Y
NW.W32.BlueCode	N	EW.W32.Bagle.fk	Y
EW.W32.Klez.e	Y	NW.W32.Bobic.k	Y
EW.W32.Bagle.h	Y	EW.W32.Klez.f	Y
NW.W32.Bozori.b	Y	EW.W32.Bagle.i	Y
EW.W32.Klez.g	Y	NW.W32.Bozori.e	Y
EW.W32.Bagle.j	Y	EW.W32.Klez.i	Y

Figure 6.14: Virus Test Results Kernel implementation of SRRAT - 1

Virus Name	*SRR* Detected	Virus Name	*SRR* Detected
NW.W32.Bozori.j	Y	EW.W32.Bagle.k	Y
EW.W32.Klez.j	Y	NW.W32.CodeGreen.a	N
EW.W32.Bagle.m	Y	EW.W32.LovGate.g	Y
NW.W32.CodeRed	N	EW.W32.Bagle.n	Y
EW.W32.Nyxem	Y	NW.W32.Cycle.a	Y
EW.W32.Bagle.o	Y	EW.W32.Merkur.b	N
NW.W32.Dipnet-f	Y	EW.W32.Bagle.q	Y
EW.W32.Mimail.j	Y	NW.W32.Dabber.b	N
EW.W32.Bandet.a	Y	EW.W32.Mydoom.ax	Y
NW.W32.Dabber.c	Y	EW.W32.Banza	N
EW.W32.Mydoom.b	N	NW.W32.Daper.a	N
EW.W32.Bater.a	Y	EW.W32.Mydoom.e	Y
NW.W32.Domwoot.c	Y	EW.W32.Benny	Y
EW.W32.Mydoom.l	Y	NW.W32.Doomjuice.b	Y
EW.W32.Bimoco.a	Y	EW.W32.Mydoom.m	Y
NW.W32.Doomjuice.d	Y	EW.W32.Blare	N
EW.W32.Mydoom.q	Y	NW.W32.Doomran	Y
EW.W32.Blitzy	N	EW.W32.NSky	N
NW.W32.Ezio.a	Y	EW.W32.Bonorm	N
EW.W32.NSky.b	Y	NW.W32.Francette.a	N
NW.W32.Francette.b	N	NW.W32.SdBoter.k	N
P2PW.W32.Cabby	Y	NW.W32.Francette.c	N
NW.W32.Shelp.a	N	P2PW.W32.Cake	N
NW.W32.Francette.d	N	NW.W32.Spoder.a	N
P2PW.W32.Carfin	N	NW.W32.Francette.e	N
NW.W32.Stap.b	Y	P2PW.W32.Cassidy	Y
NW.W32.Francette.g	N	NW.W32.Stap.e	N
P2PW.W32.Cayen.a	N	NW.W32.Hiberium.b	N
NW.W32.Stap.f	N	P2PW.W32.Cocker	Y
NW.W32.Incef.a	Y	NW.W32.Syner.a	Y
P2PW.W32.Compatex	Y	NW.W32.Incef.b	Y
NW.W32.Theals.b	N	P2PW.W32.Compux.a	Y
NW.W32.Kidala.a	Y	NW.W32.Theals.c	N
P2PW.W32.Cozit	Y	NW.W32.Kidala.b	Y
NW.W32.Vesser.a	N	P2PW.W32.Dafly.b	Y
NW.W32.Lebreat.a	Y	NW.W32.Webdav.a	Y
P2PW.W32.Dani	N	NW.W32.Lebreat.b	Y
NW.W32.Welchia.a	Y	P2PW.W32.Darby.b	N
NW.W32.Lebreat.d	Y	NW.W32.Welchia.b	Y
P2PW.W32.Darby.c	N	NW.W32.Lebreat.m	Y

Figure 6.15: Virus Test Results Kernel implementation of SRRAT - 2

Virus Name	*SRR* Detected	Virus Name	*SRR* Detected
NW.W32.Welchia.c	Y	P2PW.W32.Delf.a	Y
NW.W32.Muma.c	Y	NW.W32.Welchia.e	Y
P2PW.W32.Disager	Y	NW.W32.Maslan.b	N
NW.W32.Xatch.a	N	P2PW.W32.Druagz	Y
NW.W32.Muma.b	Y	NW.W32.Zan	N
P2PW.W32.Erdam	Y	NW.W32.Muma.c	Y
NW.W32.Zusha.a	Y	P2PW.W32.Flocker.01	Y
NW.W32.Mytob.q	Y	NW.W32.Zusha.b	Y
P2PW.W32.Franvir	N	NW.W32.Nimda	Y
NW.W32.Zusha.c	Y	P2PW.W32.Furby	N
NW.W32.OpasSoft.a.pac	N	NW.W32.Zusha.e	N
P2PW.W32.Gagse	Y	NW.W32.Padobot.m	Y
NW.W32.Zusha.f	N	P2PW.W32.Gedza.b	N
NW.W32.Protoride.aa	N	P2PW.W32.Abuva	N
P2PW.W32.Gedza.c	Y	NW.W32.Protoride.ai	N
P2PW.W32.Adil	Y	P2PW.W32.Gotorm	Y
NW.W32.Protoride.al	N	P2PW.W32.Agobot.a	Y
P2PW.W32.Grompo	Y	NW.W32.Protoride.ar	N
P2PW.W32.Agobot.b	Y	P2PW.W32.Habaku.a	N
NW.W32.Protoride.b	N	P2PW.W32.Agobot.c	Y
P2PW.W32.Habku.b	N	NW.W32.Protoride.bk	Y
P2PW.W32.Agobot.d	Y	P2PW.W32.Halfint	N
NW.W32.Protoride.e	N	P2PW.W32.Alcan.a	Y
P2PW.W32.Huntox	N	NW.W32.Protoride.f	N
P2PW.W32.AntiFizz	Y	P2PW.W32.Ident	N
NW.W32.Protoride.g	N	P2PW.W32.Aplich	Y
P2PW.W32.Ihit.a	N	NW.W32.Raleka.b	N
P2PW.W32.Apsiv	N	P2PW.W32.Insta.a	N
NW.W32.Rega.a	Y	P2PW.W32.Aritim	Y
P2PW.W32.Inter	Y	NW.W32.Salie.a	N
P2PW.W32.Ariver	Y	P2PW.W32.Irkaz	Y
NW.W32.Sasser.b	Y	P2PW.W32.Backterra.a	N
P2PW.W32.Kabak.a	N	NW.W32.Sasser.d	Y
P2PW.W32.Banuris.a	Y	P2PW.W32.Kamadina	N
NW.W32.SdBoter.a	N	P2PW.W32.Benjamin.a.exe	Y
P2PW.W32.Kamafe	N	NW.W32.SdBoter.b	Y
P2PW.W32.Bereb.a	Y	P2PW.W32.Kanyak.a	Y
NW.W32.SdBoter.c	Y	P2PW.W32.Bereb.b	Y
P2PW.W32.Kapucen.b	Y	NW.W32.SdBoter.g	Y
P2PW.W32.Blaxe	Y	P2PW.W32.Kazeus	Y

Figure 6.16: Virus Test Results Kernel implementation of SRRAT - 3

Virus Name	*SRR* Detected	Virus Name	*SRR* Detected
P2PW.W32.Kenfo	N	V.W32.Arch.a	Y
V.W32.Gipiras.a	N	P2PW.W32.Kevor	N
V.W32.Aris	N	V.W32.Gpcode.ac	N
P2PW.W32.Kifie.a	N	V.W32.Artelad.2173	N
V.W32.Halen.2618	N	P2PW.W32.Kifie.c	Y
V.W32.Bacros.a	Y	V.W32.Hezhi	N
P2PW.W32.Kifie.f	Y	V.W32.Banaw.2157	N
V.W32.Jlok	Y	P2PW.W32.Lamerx	N
V.W32.Barum.1536	N	V.W32.Kenfa.a	N
P2PW.W32.Lemb.b	Y	V.W32.Basket.a	N
V.W32.Levi.2961	N	P2PW.W32.Mantas.a	Y
V.W32.Bayan.a	N	V.W32.Mental	N
P2PW.W32.Niklas.a	Y	V.W32.BCB.a	Y
V.W32.Mental.10000	N	P2PW.W32.Niklas.b	Y
V.W32.Bee	N	V.W32.Mental.10472	N
P2PW.W32.Niklas.c	Y	V.W32.Beef.2110	N
V.W32.Neshta.a	Y	P2PW.W32.Opex.a	Y
V.W32.Bender.1363	N	V.W32.Nlip	N
P2PW.W32.Polipos	N	V.W32.Bika.1906	N
V.W32.Niya.a	N	P2PW.W32.SpyBot	N
V.W32.Blateroz	N	V.W32.Parite.a	Y
P2PW.W32.Vagas.a	Y	V.W32.Bluback.1376	N
V.W32.Parite.b	Y	P2PW.W32.Walrain	Y
V.W32.Blueballs.4117	N	V.W32.Porad.a	N
P2PW.W32.Weakas	Y	V.W32.Bogus.4096	N
V.W32.Redemption	N	P2PW.W32.Zaka.a	Y
V.W32.Bondage.968.a	N	V.W32.Savior.1680	N
P2PW.W32.Zaka.f	Y	V.W32.Butter	N
V.W32.Seppuku.6834	Y	P2PW.W32.Zaka.m	Y
V.W32.Cabanas.e	N	V.W32.Sinco	N
V.W32.Small.a	N	V.W32.CabInfector	N
V.W32.Canbis.a	Y	V.W32.Small.c	N
V.W32.Spreder	Y	V.W32.Cecile	N
V.W32.Civut.a	N	V.W32.Storm	N
V.W32.Storm-2	N	V.Boot-DOS.Tequila	N
V.W32.Cmay.1222	N	V.W32.Stream.a	N
V.DOS.Maltese-Amoeba.2367	N	V.W32.Cornad	N
V.W32.Cloz.a	N		

Figure 6.17: Virus Test Results Kernel implementation of SRRAT - 4

Virus Name	*SRR* Detected	Virus Name	*SRR* Detected
V.W32.Stream.b	N	V.DOS.OneHalf.3666	N
V.W32.Crosser	N	V.W32.Sugin	N
V.MSIL.Gastropod	N	V.W32.Crypto	N
V.W32.Tapan.3882	N	V.MSWord.Blaster	N
V.W32.CTX.6886	N	V.W32.TeddyBear	N
V.MSWord.Melissa	N	V.W32.Delfer.a	Y
V.W32.Tenga.a	Y	V.VBS.Lucky2	N
V.W32.Devir	N	V.W32.Teta.a	Y
V.VBS.H	Y	V.W32.Dictator.2304	N
V.W32.Thorin.11932	N	V.W32.Aidlot	N
V.W32.Dislex	N	V.W32.Thorin.b	N
V.W32.Aldebaran.8365.a	N	V.W32.Donut	Y
V.W32.Thorin.c	N	V.W32.Aldebaran.8365.b	N
V.W32.Elkern.a	N	V.W32.Thorin.d	N
V.W32.Andras.7300	N	V.W32.Emotion.a	N
V.W32.Thorin.e	N	V.W32.Apathy.5378	Y
V.W32.Enumiacs.6656	N	V.W32.VChain	N
V.W32.Apoc.a	N	V.W32.Fosforo	N
V.W32.Voltage.A	N	V.W32.Apparition	N
V.W32.Ghost.1667	N	V.W32.Watcher.a	Y
V.W32.Xorala	N	V.W32.Xorala.b	N
V.W32.Yerg.9571	N	V.W32.Younga.4434	Y
V.W32.Zevity	N	V.W32.ZMist	N
V.W32.Zori.a	Y	V.W32.ZPerm.b	N
V.W32.ZPerm.b2	N	V.Win9x.CIH	N
V.Win9x.DarkSide.1371	N	V.Win9x.Sledge.735.b	N
V.Win9x.Small.140	N	Worm.VB-16	Y

Figure 6.18: Virus Test Results Kernel implementation of SRRAT - 5

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

This dissertation has address the problem of finding a characteristic present in all viruses that is suitable to detect both known and unknown viruses belonging to different categories and under several execution conditions. The problems of current virus detection both signature and behavior based are well established in the literature. This research presents a behavior based approach to virus detection based on *self-reference replication* (SR-replication) which is a characteristic assumed to be unique to viruses and not commonly occurring in benign processes. Using this characteristic a behavior based detection approach was formalized using directed call graphs to detect the self replication of a virus file to some other preexisting or newly created file on a system. The approach works without any a priori knowledge of previously discovered viruses. The approach simply detects attempts by a process to perform SR-replication. The approach was tested by conducting static analysis of known viruses and benign processes.

The results showed SR-replication to be occurring in most viruses and in none of the tested benign processes. Two implementations of the approach were created and tested. In both cases no false positives were produced. There were false negatives which resulted from the implementations lacking needed functionality to cover more sophisticated viruses capable of executing and avoiding detection. The overall conclusions of this dissertation is twofold: First, SR-replication can be used as a characteristic to differentiate between viruses and benign processes. Furthermore SR-replication can detect known and unknown viruses when they execute without

any a priori knowledge. This ability makes SR-replication well suited to detect newly released unknown file infecting viruses upon their initial attempts to executed SR-replication on a system. Second, a real time process monitor and virus detector on a system can be implemented and is usable using SR-replication as long as the implementation is at a low level of the computer system, for example the in Kernel mode.

A final observation about SR-replication follows: this dissertation has shown the virus detection abilities of SR-replication to be very good with strong potential but in its current state, this form of virus detection is not suitable as a stand alone solution for all forms of viruses detection. From analyzing the test results it is clear that SR-replication, in its current form, is best used in conjunction with other forms of virus detection to provide a robust compound solution that is better suited to face the issues of virus attacks on computer systems with its strongest novel contribution being the ability to detect newly released not yet analyzed viruses initially attempting SR-replication on preexisting or newly created files on a computer system.

7.2 Future Work

The approach of SR-replication presented here can be strengthened by expanding it to detect replication of viruses into memory and not just files. Another key aspect is to extend this approach to detect SR-replication across a network. Another form of replication called *indirect self-reference replication* (ISR-replication) was briefly discussed. This is a whole new field of replication complimentary to SR-replication. Creating algorithms to detect ISR-replication will provide an overall more robust approach to detecting virus replication in general with no a priori knowledge of any known viruses. This is essential to protect computer systems from future virus attacks. The problems faced during creation of our implementation of SRRAT need to be addressed with an appropriate solution. Specifically the memory problems found

in the Kernel implementation of SRRAT need to be solved to provide a complete implementation that works the same as the user implementation of SRRAT building directed call graphs to determine if SR-replication has occurred in real time and provide the functionality of terminating a process when it does exhibit SR-replication.

BIBLIOGRAPHY

[1] Conry-Murray A. Behavior blocking stops unknown malicious code. *Network Magazine*, June 2002. http://www.networkmagazine.com.

[2] Gostev A. Malicious code evolution: July to september 2007. http://www.viruslist.com/en/analysis?pubid=204791973.

[3] Gostev A. Malware evolution: January to july 2007. http://www.viruslist.com/en/analysis?pubid=204791966.

[4] Gostev A. Kaspersky security bulletin 2006: Malware evolution. *Viruslist.com*, February 2007.

[5] Morales J. A., Clarke P. J., Kibria B.M., and Deng Y. Testing and evaluating virus detectors for handheld devices. *Journal in Computer Virology*, September 2006.

[6] Anarch email worm. http://www.viruslist.com/en/viruses/encyclopedia.

[7] Api spy 32. http://www.matcode.com/apis32.htm (November 2007).

[8] Livingston B. How long must you wait for an anti-virus fix? *Datamation*, February 2004. http://itmanagement.earthweb.com/.

[9] Nachenberg C. Behavior blocking: The next step in anti-virus protection. *Security Focus*, March 2002. http://www.securityfocus.com/infocus/1557.

[10] Langston C.G. Self reproduction in cellular automata. *Physica D*, 10:135–144, 1984.

[11] Denning D. Cyberterrorism testimony before the special oversight panel of terrorism committee on armed services, house of representatives, May 2000. http://www.cs.georgetown.edu/ denning/infosec/cyberterror.html.

[12] Mutz D., Valeur F., Vigna G, and Kruegel C. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, 2006.

[13] Bell D.E. and LaPadula L.J. Secure computer systems: Mathematical foundations and model. Technical report, The Mitre Corporation, 1973.

[14] Filiol E. *Computer Viruses: from Theory to Applications*. IRIS International series, Springer Verlag, 2005. ISBN 2-287-23939-1.

[15] Filiol E. Malware pattern scanning schemes secure against black box analysis. In *European Institute for Computer Anti-Virus Research (EICAR)*, 2006.

[16] Kaspersky E. Problems for av vendors: some thoughts. *Virus Bulletin*, April 2006.

[17] Messmer E. Behavior blocking repels new viruses. *Network World Fusion*, January 2002. http://www.nwfusion.com/news/2002/0128antivirus.html.

[18] Codd E.F. *Cellular Automata*. Academic Press, 1968.

[19] Daniel R. Ellis, John G. Aiken, Kira S. Attwood, and Scott D. Tenaglia. A behavioral approach to worm detection. In *WORM '04: Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 43–53, New York, NY, USA, 2004. ACM Press.

[20] Cohen F. *Computer Viruses*. PhD thesis, University of Southern California, 1986.

[21] Cohen F. Computer viruses - theory and experiments. *Computers and Security*, 6:22–35, 1987.

[22] Cohen F. *A Short Course on Computer Viruses*. Wiley Professional Computing, 1994. ISBN 0-471-00769-2.

[23] Schneider F. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000. ACM Press, New York, NY, USA. http://doi.acm.org/10.1145/353323.353382.

[24] D. Golden and M. Pechura. The structure of microcomputer file systems. *Commun. ACM*, 29(3):222–230, 1986.

[25] G. Hoglund and J. Butler. *Rootkits: subverting the Windows Kernel*. Addison Wesley Professional, 2005.

[26] Bergeron J., Debbabi M., Erhioui M. M., and Ktari B. Static analysis of binary code to isolate malicious behaviors. In *Proceedings of the IEEE 4th International Workshops on Enterprise Security (WETICE'99),*, June 1999.

[27] Byl J. Self reproduction in cellular automata. *Physica D*, 34:295–299, 1989.

[28] Evers J. Computer crimes cost 67 billion, fbi says. *cnet News.com*, January 2006.

[29] Von Neumann J. Theory of self-reproducing automata. Technical report, University of Illinois, 1966.

[30] Morales J.A., Clarke P.J., and Deng Y. Characterizing virus replication. 2nd International Workshop on the Theory of Computer Viruses in Nancy, France, May 2007.

[31] Anderson J.P. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., April 1980.

[32] Kornakov K. Cybersecurity on the agenda for us government. http://www.viruslist.com/en/news?id=208274036.

[33] Kaspersky anti-virus. http://www.kaspersky.com.

[34] Biba K.J. Integrity considerations for secure computer systems. Technical report, USAF Electronic Systems Division, 1977.

[35] T.A. Linden. Operating system structures to support security and reliable software. *ACM Comput. Surv.*, 8(4):409–445, 1976.

[36] Adleman L.M. An abstract theory of computer viruses. In *CRYPTO '88: Advances in Cryptology*, pages 354–374. Springer, 1988.

[37] Bernaschi M., Gabrielli E., and Mancini L.V. Remus: a security-enhanced operating system. *ACM Trans. Inf. Syst. Secur.*, 5(1):36–61, 2002.

[38] Christodorescu M. and Jha S. Testing malware detectors. *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 34–44, 2004. ACM Press, New York, NY, USA. http://doi.acm.org/10.1145/1007512.1007518.

[39] Christodorescu M., Jha S., Maughan D., Song D., and Wang C., editors. *Malware Detection*. Springer, 2007.

[40] Ludwig M.A. *Computer Viruses and Artificial Life and Evoluiton*. American Eagle Press, 1993.

[41] G. Nebbett. *Windows NT/2000 Native API Reference*. Macmillan Technical Publishing, 2000.

[42] Offensive computing malware repository. http://www.offensivecomputing.net.

[43] Singh P. and Lakhotia A. Analysis and detection of computer viruses and worms: an annotated bibliography. In *ACM SIGPLAN Notices*, volume 37, pages 29–35, 2002. ACM Press, New York, NY, USA. http://doi.acm.org/10.1145/568600.568608.

[44] Szor P. *The Art of Computer Virus Research and Defense*. Symantec Press and Addison-Wesley, 2005. ISBN 9-780321-304544.

[45] Sekar R., Bendre M., Dhurjati D., and Bollineni P. A fast automaton-based method for detecting anomalous program behaviors. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 144, Washington, DC, USA, 2001. IEEE Computer Society.

[46] Rega network worm. http://www.viruslist.com/en/viruses/encyclopedia.

[47] Forrest S., Hofmeyr S.A., Somayaji A., and Longstaff T.A. A sense of self for unix processes. In *Proceedings of 1996 IEEE Symposium on Computer Security and Privacy*, 1996.

[48] Gordon S. and Ford R. Computer crime revisited: The evolution of definition and classification. In *European Institute for Computer Anti-Virus Research (EICAR)*, 2006.

[49] Moore S. The spread of the witty worm. http://www.caida.org/analysis/security/witty/.

[50] Kleene S.C. On notation for ordinal numbers. *Journal of Symbolic Logic*, 3:150–155, 1938.

[51] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[52] V. Skormin, A. Volynkin, D. Summerville, and J. Moronski. Prevention of information attacks by run-time detection of self-replication in computer codes. *Journal of Computer Security*, 15(2):273–302, 2007.

[53] Chari S.N. and Cheng P. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Trans. Inf. Syst. Secur.*, 6(2):173–200, 2003.

[54] Symantec antivirus research center. http://securityresponse.symantec.com/.

[55] Microsoft sysinternals software. http://www.microsoft.com/technet/sysinternals/.

[56] Bradley T. The new virus fighters. *Datamation*, January 2006. http://www.pcworld.com/article/id,124163-page,4/article.html.

[57] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[58] R. Vieler. *Professional Rootkits*. Wrox Press, 2007.

[59] Vmware virtual workstation. http://www.vmware.com.

[60] Vx heavens. http://vx.netlux.org/.

[61] Burks A. W. Essays on cellular automata. Technical report, University of Illinois, 1970.

[62] Wikipedia.com. http://www.wikipedia.com/.

[63] Windows api reference. http://msdn2.microsoft.com/en-us/library/aa383749.aspx.

VITA

JOSE ANDRE MORALES

1997	B.S., Computer Science Florida International University Miami, Florida
2004	M.Sc., Computer Science Florida International University Miami, Florida
2008	Doctoral Candidate in Computer Science Florida International University Miami, Florida

PUBLICATIONS AND PRESENTATIONS

Morales J.A., Clarke P.J., Kibria B.M.G. and Deng Y. (2008). Characterization of Virus Replication, in *Journal in Computer Virology Special Issue on Theory of Computer Viruses Workshop*, Springer-Verlag.

Morales J.A., Clarke P.J., Kibria B.M.G. and Deng Y. (2006). Testing and Evaluating Virus Detectors for Handheld Devices, in *Journal in Computer Virology Special Issue on Mobile Malware and Anti-malware Technologies*, Springer Paris, Volume 2/Number 2, pg. 135-147.

Morales J.A., Clarke P.J., and Deng Y. (2008). Detecting Self-Reference Virus Replication, in *Proceedings of the 17th Annual European Institute for Computer Anti-Virus Research (EICAR) Conference*, May 3-8, Laval France.

Morales J.A., Clarke P.J., and Deng Y. (2008). Characterizing and Detecting Virus Replication, in *Proceedings of the Third International Conference on Systems (ICONS)*, April 13-18 2008, Cancun Mexico.